Escuela Politécnica Superior

Programación de Aplicaciones OpenCV sobre Sistemas Heterogéneos SoC-FPGA

Máster Universitario en Ingeniería de Telecomunicación

Trabajo Fin de Máster

Autor:
Francisco José Sanchis Cases

Tutor/es:
Dr. Sergio Cuenca Asensi

Dr. Antonio Martínez Álvarez

Diciembre 2014

Universitat d'Alacant
Universidad de Alicante

Universidad De Alicante

Escuela Politécnica Superior

Máster Universitario en Ingeniería de Telecomunicación

Universitat d'Alacant
Universidad de Alicante

trabajo final de máster:

*P*rogramación de *A*plicaciones *O*pen*CV* sobre *S*istemas *H*eterogéneos *S*o*C*-*FPGA*

Autor:
Francisco Jose Sanchis Cases

Directores:
Dr. Sergio Cuenca Asensi

Dr. Antonio Martínez Álvarez

DICIEMBRE DE 2014

Agradecimientos

Antes de todo, quiero agradecer la oportunidad que me han otorgado el Dr. Sergio Cuenca Asensi y el Dr. Antonio Martínez Álvarez para realizar el Proyecto Final de Máster en el grupo *UNICAD* del departamento *DTIC* (Departamento de Tecnología Informática y Computación) de la Universidad de Alicante. Gracias a ellos, he podido mejorar mis aptitudes en el campo de visión artificial, en el uso de la biblioteca *OpenCV* y en los sistemas empotrados como las nuevas *FPGAs* con ARM.

Además, quiero agradecer el apoyo otorgado por mis compañeros de laboratorio y compañeros de clase, ya que en algunos momentos del transcurso del proyecto, me han ayudado en algunos puntos que me resultaban un poco más complicados. También agradecer a mi familia por la ayuda y apoyo aportado en todo este año en la realización del máster.

Y por último dar un agradecimiento a aquellas personas que me han apoyado en mi etapa por la *Universidad de Alicante*.

Índice general

Lista de Figuras

Lista de Tablas

Glosario

CPU Central Processing Unit. 1, 3, 22, 23

DMA Acceso Directo a Memoria. 21

DSE Exploración del Espacio de Diseño. 7

ESL Electronic System-Level. 6

FIFO First In First On. 11, 24

FPGA Field Programmable Gate Array. 1, 2, 4, 10, 13, 29

FSMDs Finite-State Machines with Datapath. 6

GP-GPU General-Purpose Computing on Graphics Processing Units. 3

GPU Graphics Processing Unit. 1–4

HLL High-level languages. 1

HLS High-level synthesis. 6, 11

JIT Just-in-Time. 3

NAC N-Address Code. 6

PLL Phase-Locked Loop. 21

RTL Register-Transfer Level. 6

SLAM Simultaneous Localization And Mapping. 1

SoC System on Chip. 1, 2

VGA Video Graphics Array. 18

VHDL Very High Speed Integrated Circuit. 6

Capítulo 1

Introducción

Este proyecto llamado *Programación de Aplicaciones OpenCV sobre Sistemas Heterogéneos SoC-FPGA* plantea una nueva forma sencilla de diseñar aplicaciones en el campo de *Visión Artificial* para las nuevas plataformas heterogéneas *SoC-FPGA*, que se basa en un conjunto de herramientas de síntesis SW y HW propias y de terceros.

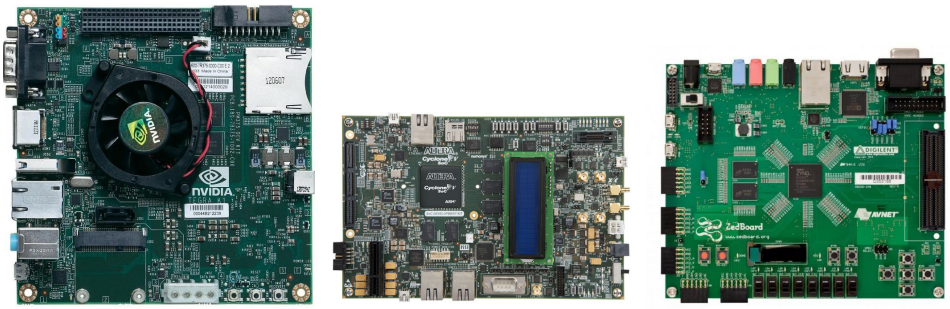
El trabajo se ha desarrollado dentro de un proyecto *AVANZA* titulado *PRADVEA* visión y robótica autónoma: *rovers* y vehículos aéreos, liderado por la empresa *IXION Industry and Aerospace*. Dicho proyecto se centra en la evaluación de herramientas de diseño de Field Programmable Gate Array (FPGA) basadas en High-level languages (HLL), en el desarrollo de bibliotecas de procesamiento de imagen y en la implementación de un caso de estudio (Simultaneous Localization And Mapping (SLAM)).

El proyecto se ha realizado en el seno del grupo *UniCAD* [1], de la Universidad de Alicante. Dicho grupo es un grupo joven formado por profesores de los *Departamentos de Tecnología Informática y Computación* y *Ciencia de la Computación e Inteligencia Artificial* de la *Universidad de Alicante* que posee gran experiencia en estos campos.

1.1. Motivación

Las últimas tendencias tecnológicas relacionadas con los (System on Chip (SoC)) están permitiendo la continua aparición de nuevas plataformas cada vez más heterogéneas, no sólo por la cantidad de módulos digitales o analógicos que integran, sino también por la posibilidad de combinar distintos modelos de computación cada vez más complejos. Estos nuevos dispositivos se basan principalmente en tres componentes: Central Processing Unit (CPU), FPGA y Graphics Processing Unit (GPU), lo que ofrece al ingeniero múltiples alternativas de diseño.

Como ejemplo de SoC basado en GPU se destaca la plataforma *Jetson TK1* de *nVidia* [2], un dispositivo que consta de un procesador *ARM Cortex A-15* de cuatro núcleos que integra una *GPU Kepler* de 192 núcleos. Por otra parte, como ejemplo de SoC basado en FPGA se destaca la familia *Zynq* de *Xilinx* [3], que integra en una FPGA de la serie 7 dos procesadores *ARM Cortex A-9* a 1GHz, o la familia *Cyclone* V *SoC* de bajo consumo de *Altera* [4] que consta de dos núcleos *ARM Cortex A-9* a 925MHz. Como se puede apreciar, la mayoría de estas plataformas integran núcleos *ARM* de última generación, muy extendidos en el campo de la telefonía móvil y sistemas empotrados en general.



(a) Jetson TK1 (b) Cyclone V SoC (c) Zedboard

Figura 1.1: Ejemplo de tres plataformas heterogéneas actuales

La incorporación de arquitecturas especializadas a estos *SoCs*, permite abordar aplicaciones de alta demanda computacional en diversos campos: como la visión por computador [5], el tratamiento digital de señales [6] o las redes de comunicación [7]. Además, estos nuevos dispositivos son en muchas ocasiones los candidatos perfectos gracias a su miniaturización, su reducido coste y su consumo de potencia.

Las plataformas *SoC-FPGA* permiten diseñar desde cero un sistema hardware que coopere con el resto del sistema, sin embargo, este tipo de diseño requiere de habilidades especiales tanto en el diseño Hw como en el Sw.

Por otra parte, las bibliotecas software para visión artificial, o destinadas a procesamiento de visión proporcionan un gran recurso para el diseño de aplicaciones para las plataformas tradicionales (por ejemplo x86 o AMD64). Con dichas bibliotecas se puede diseñar una aplicación y comprobar su funcionalidad en un sistema tradicional antes de migrar el diseño a una plataforma heterogénea. Entre sus características destacamos las siguientes.

- Están implementadas en un lenguaje estándar. Típicamente C o C++.

- Poseen entornos de desarrollo especializados.
- Admiten ampliaciones de nuevos componentes por la comunidad.
- Son de código abierto.

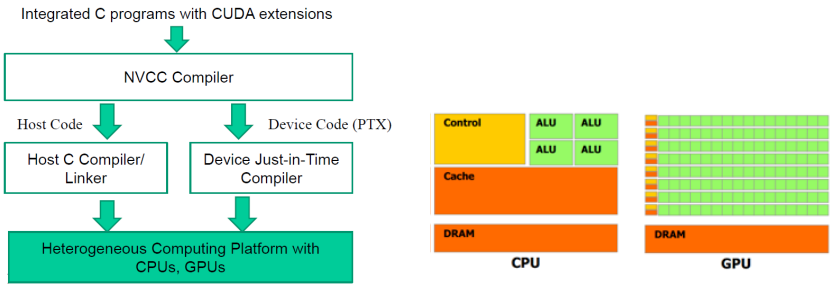
Entre las bibliotecas que cumplen estas características, destaca *OpenCV* [8]. Dicha biblioteca es gratuita (basada en una licencia de BSD), de código abierto y contiene más de 2500 funciones de procesamiento de vídeo. A parte de esto, *OpenCV* posee un gran interés en incorporar las nuevas plataformas embebidas a la biblioteca como los dispositivos móviles y es ampliamente utilizada en el ámbito académico e industrial.

Al unificar los dos mundos, las bibliotecas software y las plataformas heterogéneas, se observan unos problemas o dificultades que hay que resolver a la hora de plantear un diseño heterogéneo. Dichos problemas son:

- La complejidad de diseñar es mayor debido a que es necesario trabajar con distintos tipos de arquitecturas y, cada una, con distintas metodologías de programación.
- El tiempo de desarrollo aumenta al considerar las tareas de integración, comunicación entre arquitecturas, etc.
- Las optimizaciones se hacen difíciles por que no sólo tenemos que considerar las arquitecturas por separado sino también las interacciones entre ambas.
- La verificación del sistemas en un todo necesita de metodologías y herramientas complejas.

Seleccionado las metodologías de diseño empleadas por los principales fabricantes de estas plataformas se pueden agrupar en dos enfoques claramente diferenciados.

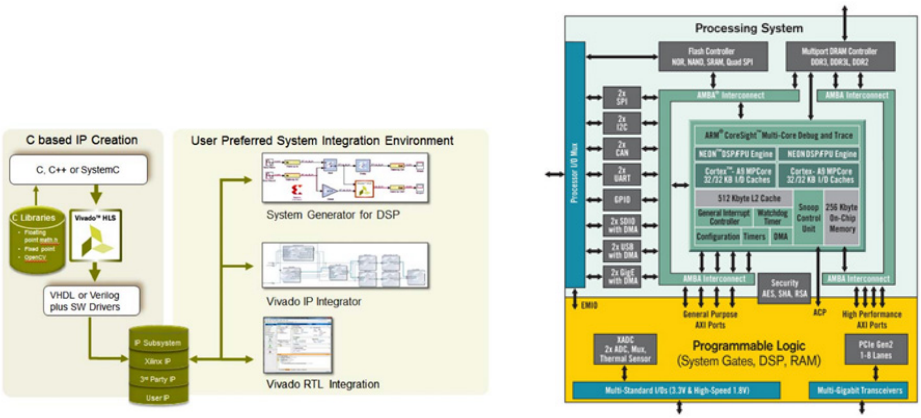
El primero de ellos es el tomado por los fabricantes de General-Purpose Computing on Graphics Processing Units (GP-GPU) o, más concretamente, el enfoque que toma *Nvidia* con su biblioteca *CUDA*. Dicho enfoque, centrado en compilador, se describe en la figura 1.2(a) donde a partir de un diseño descrito en C/C++ y extensiones *CUDA* se genera un sistema acelerado por Hw gracias al compilador *NVCC*. El compilador *NVCC* genera automáticamente un fichero objeto para la CPU (host) y un código para la GPU que es interpretado al vuelo (Just-in-Time (JIT)) por el driver del dispositivo. La característica fundamental de estas plataformas es poseer una arquitectura fija como se puede observar en la gráfica 1.2(b), así como los canales y procedimientos predefinidos para la transferencia de datos y control.



(a) Compilador (b) Arquitectura GPU

Figura 1.2: Enfoque GP-GPU de Nvidia CUDA basada en compilador

El segundo enfoque, centrado en IPs, es el empleado por los fabricantes de FPGAs o, en está descripción, el tomado por *Xilinx*. Este enfoque se basa en la generación de núcleos Hw (IPs) a partir de un diseño a alto nivel 1.3(a), es decir, un diseño Hw descrito en C++ con directivas de síntesis. El núcleo sintetizado por el compilador es instanciado en unas infraestructuras Hw la cual se diseña para una plataforma concreta (ejemplo la familia *Zynq* 1.3(b)). Para la ejecución del diseño es necesario generar una infraestructura Sw de control que se encarga de gestionar el sistema Hw completo.



(a) Xilinx IP (b) Arquitectura Zynq

Figura 1.3: Enfoque FPGA de Xilinx basada en IP

La diferencia más destacada entre las GPU y las FPGA es la arquitectura. Como antes se mencionó, la GPU posee una arquitectura fija pero en la FPGA es variable y, por ello, proporciona un nivel mayor de posibilidades de diseño. Por contra, esta infinidad de posibilidades implica un nivel de dificultad mayor a la hora de diseñar.

El enfoque que este trabajo quiere acometer es el centrado en compilador, que será el encargado de dirigir el co-diseño de un sistema de visión para las plataformas *FPGAs* con *ARM*.

Además, se pretende dar unas soluciones a los problemas descritos anteriormente con:

- Como método de especificación de sistemas de procesamiento de visión, se plantea una descripción SW. Esto permitirá utilizar una metodología conocida y asequible.
- El tiempo en diseñar se reduce gracias a generación automática a partir de información proporcionado por el usuario.
- Las optimizaciones se abordaran a varios niveles y de forma automática.
- La verificación se plantea a alto nivel y con la utilización de arquitecturas Hw/Sw de referencia previamente verificadas.

1.2. Objetivo

El objetivo del presente trabajo se resume en el desarrollo de un *Entorno de Diseño* amigable para la implementación de sistemas de Visión Artificial partiendo de proyectos software descritos mediante código *C/C++* con la biblioteca *OpenCV* y para la plataforma *Zynq* de *Xilinx*.

Este entorno debe ser capaz de:

- Generar una infraestructura Hw de la parte del procesamiento que se pretende acelerar.
- Generar un diseño completo Hw/Sw implementable para la plataforma heterogénea deseada.
- Verificar el sistemas a varios niveles: Test Sw funcional, uso de arquitecturas verificadas, etc.

1.3. Estado del Arte

En la actualidad se puede encontrar una gran variedad de herramientas que facilitan el diseño Hw/Sw o co-diseño. Entre ellas, las nuevas herramientas basadas en compilador *C-to-Hardware*, que permiten diseñar sistemas

Hw a partir de su especificación en lenguajes de alto nivel. Estas herramientas se están afianzándose cada vez más como una tendencia de diseño muy prometedora y cada vez más extendida.

Entre las herramientas de síntesis Hw más destacadas en este ámbito encontramos: *Hercules* [9], que a partir de un código descrito en *ANSI C* es capaz de generar automáticamente *Register-Transfer Level (RTL) VHDL*, *SystemCoDesigner* [10], capaz de generar síntesis Hw a partir de *SystemC*, y *Vivado HLS* [11] de *Xilinx*, que genera un *core Hw* a partir de un proyecto descrito en *C++*.

(a) Hercules (b) SystemCoDesigner (c) Vivado HLS

Figura 1.4: Herramientas C-to-Hardware.

`Hercules` es una herramienta de síntesis de alto nivel (High-level synthesis (HLS)) que genera automáticamente código RTL para hardware no programable. Primero se traducen programas descritos en *ANSI C* a N-Address Code (NAC), después lo convierte a Finite-State Machines with Datapath (FSMDs) extendidas (Finite-State Machines with Datapath) y, finalmente, se genera RTL VHDL. El sistema consta básicamente de dos componentes: un *frontend* (nac2cdfg) y un *backend* (cdfg2hdl)

- `nac2cdfg`: traduce de *NAC* (N-Address Code) IR a *CDFGs* (gráfico dirigido - Graphviz).
- `cdfg2hdl`: encargado de generar automáticamente la salida RTL y verificar a partir del gráfico dirigido.

También, el código generado Very High Speed Integrated Circuit (VHDL) puede ser simulado con *GHDL* (simulador VHDL) o por la herramienta *Modelsim*.

`SystemCoDesigner` es una herramienta Electronic System-Level (ESL) desarrollado en la Universidad de Erlangen-Nuremberg, Alemania. A partir de un modelo descrito en SystemC, se genera un hardware acelerado automáticamente utilizando `Forte Cynthesizer`. Esta herramienta es capaz

de realizar una Exploración del Espacio de Diseño (DSE) automáticamente y visualizar las soluciones de cada partición de la exploración. El sistema generado es basa en una arquitectura *MicroBlaze* y un Sw para esta arquitectura.

Por último, la herramienta `VIVADO HLS` de *Xilinx* que genera *cores HW* a partir de un sistema descrito en C/C++ con directivas al compilador *c-to-hardware* de *VIVADO HLS*. El sistema generado puede ser verificado funcionalmente a alto nivel. A partir del núcleo generado se tiene que utilizar otras herramientas para generar el sistema completo como son las herramientas *ISE* o *Vivado* de *Xilinx*.

Por otra parte, también han surgido nuevas herramientas de síntesis automática Hw/Sw entre las que destacan: `A2B` [12], herramienta semiautomática que genera un sistema Hw/Sw a partir de un código *C* con directivas *OpenMP* y una descripción Hw descrita en *XML*, `OmpSs` [13, 14], que genera un proyecto Sw y otro Hw a partir de un código *C* con *pragmas* propios, y necesita de herramientas de terceros como *Vivado HLS* para generar el Hw, `LegUp` [15], de código abierto y capaz de generar una descripción hardware *Verilog* a partir de un código en *C* y, por último, `MAMPSx`, [16] que genera un sistema completo Hw/Sw a partir de un código descrito en C.

Ninguna de estas herramientas está enfocada en el campo de visión por computador, por eso, el entorno propuesto está centrado en este campo y quiere proporcionar soluciones similares a las proporcionadas por las herramientas tradicionales de visión.

Capítulo 2

Propuesta de entorno de diseño

La propuesta de este trabajo es la realización de un entorno de diseño, denominado *AMAZynq*, basado en el compilador *Clang* integrado en la infraestructura de soporte para compiladores *LLVM* y en las nuevas herramientas de síntesis de alto nivel *Vivado HLS* de *Xilinx*. También, se propone una nueva metodología de exploración del espacio de co-diseño de los sistemas empotrados de procesamiento de visión en tiempo real.

Este entorno otorga al usuario inexperto una facilidad de diseñar y sintetizar sistemas hardware/software de visión y al usuario experto una reducción considerable en el tiempo de trabajo. Los resultados espacio-temporales son similares a los obtenidos mediante el diseño clásico de sistemas digitales.

2.1. Entorno de desarrollo

AMAZynq es un entorno de trabajo diseñado para la implementación de sistemas de procesamiento de visión en dispositivos *SoC-FPGA* de la familia *Zynq* de *Xilinx*. El entorno parte de una especificación de un sistema de visión descrito en *C++* y *OpenCV* y genera distintas particiones hardware y software para una arquitectura de referencia basada en la familia *Zynq* de *Xilinx*.

Los chips de la familia *Zynq* están divididos en dos bloques: el primero, un *SoC ARM Cortex A-9* de doble núcleo, un controlador de memoria y diversos periféricos, y el segundo, una *FPGA* de la serie 7 de *Xilinx* ligada estrechamente al *SoC*. La comunicación entre las dos partes se realiza mediante interfaces *AXI* [17] con un ancho de banda capaz de soportar hasta ocho flujos *Full-HD* simultáneos.

Para la generación completa del sistema, el entorno posee una dependencia con las siguientes herramientas de terceros:

- *Xilinx Vivado HLS*: Utilizada para la generación del *core Hw*.
- *Xilinx EDK*: Utilizada para la generación del archivo de configuración de la FPGA.
- *Xilinx SDK*: Herramienta para la generación del ejecutable compatible con la arquitectura ARM.

El entorno de diseño está dividido en varias bloques fundamentales (ver figura 2.1) que son independientes entre sí. Estos bloques se desarrollan de forma paralela y admiten futuras ampliaciones. Este planteamiento se debe a la continua actualización de la biblioteca *OpenCV* con nuevos métodos que exigen adaptaciones periódicas o por cambio de versión de la biblioteca *HLS* de *Xilinx*.

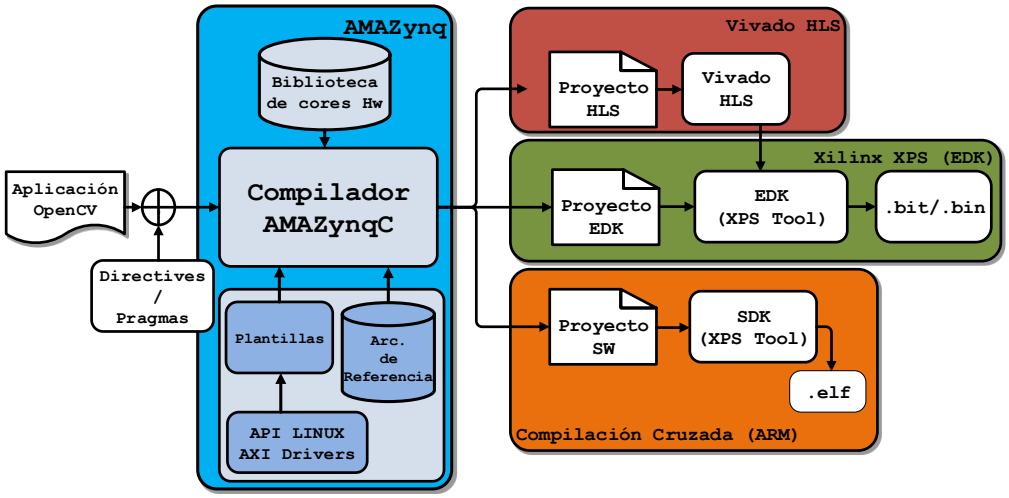


Figura 2.1: Diagrama de Bloques de AMAzynq

AMAzynq está constituido por tres módulos principales: un compilador que dirige el proceso de co-diseño de un sistema de procesamiento de vídeo especificado en alto nivel, una biblioteca de *cores* hardware basada en la biblioteca *HLS OpenCV* de *Xilinx*, y un conjunto de arquitecturas y bibliotecas software de referencia sobre las que el compilador proyectará el diseño.

2.1.1. Biblioteca de cores Hw

Este módulo está formado por una biblioteca de núcleos IPs que clona la funcionalidad de las primitivas de *OpenCV* haciendo uso de la biblioteca *HLS OpenCV* de *Xilinx*. Dicho bloque consta de todas las definiciones necesarias para diseñar un *core*, como: filtros, operadores, estructuras, etc. Además, permite trabajar con diferentes tamaños de palabra de una forma transparente.

La biblioteca clonada se ha estructurado en dos espacios de nombres: *hls* y *vs* (videostream). El primer nombre de espacios (*hls*) es el nombre usado por *Xilinx*, es debido, a que se ha extendido la biblioteca de *Xilinx* con nuevos filtros, datos, etc. El segundo, *vs*, se utiliza para diseñar sistemas con una comunicación con flujos (*streams*) de vídeo. Este bloque de la biblioteca no hace uso de la biblioteca *HLS OpenCV*, sino utiliza las bibliotecas *Bit Accurate* y *Stream* de *Vivado*, las cuales definen el tamaño, el tipo de datos y las interfaces del tipo First In First On (FIFO) (véase figura 2.2).

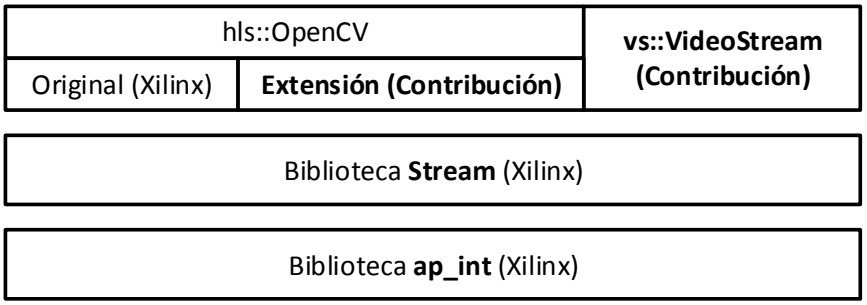


Figura 2.2: Esquema de dependencias de las bibliotecas del sistema

Estructuras Básicas:

El primer punto a describir son las estructuras básicas que se utilizan en la biblioteca *OpenCV* y en la biblioteca clonada como contenedores de una imagen. En *OpenCV* éstas estructuras son `IplImage` y `Mat`, las cuales no se soportan en síntesis hardware. Para solucionar dicho problema, la biblioteca *HLS_OpenCV* proporciona unos prototipos de funciones para convertir dichas estructuras de datos a unas similares y sintetizables por las herramientas HLS (*hls::Mat*).

En la tabla 2.1 se observa las estructuras básicos y convertidores. La

biblioteca *HLS* no contiene una estructura equivalente a `IplImage` pero si contiene métodos para convertir esa estructura de datos a `hls::Mat` y viceversa.

Tabla 2.1: Métodos y Estructuras Básicas de *OpenCV* y de *HLS_OpenCV*

Estructuras Básicas:		**Convertidores:**	
OpenCV	**HLS**	**OpenCV→HLS**	**Conversión HLS→OpenCV**
cv::IplImage	-	IplImage2hlsMat()	hlsMat2IplImage()
cv::Mat	hls::Mat	cvMat2hlsMat()	hlsMat2cvMat()

Debido a lo anterior, la única estructura sintetizable es `hls::Mat`. Dicha clase se define mediante el prototipo de *C++* (véase figura 2.3) donde los parámetros de la plantilla son el tamaño de filas (`ROWS`) y columnas (`COLS`) de la imagen y el tipo de dato del píxel (`T`).

```
template<int ROWS, int COLS, int T> // T: type of Mat
class Mat {
public:
   Mat();      // default constructor
   Mat(int _rows, int _cols);
   Mat(Size _sz);
   void init(int _rows, int _cols);
   void assignto(Mat<ROWS, COLS, T>& mat);
   Scalar<HLS_MAT_CN(T), HLS_TNAME(T)> read();
   void read(Scalar<HLS_MAT_CN(T), HLS_TNAME(T)>& s);
   void write(Scalar<HLS_MAT_CN(T), HLS_TNAME(T)> s);
   void operator >> (Scalar<HLS_MAT_CN(T), HLS_TNAME(T)>& s);
   void operator << (Scalar<HLS_MAT_CN(T), HLS_TNAME(T)> s);
   bool empty();
   const int type() const;     // return matrix type
   const int depth() const;    // return matrix depth
   const int channels() const; // return matrix channels
   HLS_SIZE_T rows, cols;      // actual image size
   hls::stream<HLS_TNAME(T)> data_stream[HLS_MAT_CN(T)];
        // image data stream array
};
```

Figura 2.3: Plantilla de la clase `hls::Mat` clonada de *OpenCV*

Además, esta clase incluye métodos que permiten realizar lecturas y escrituras (`read()` o `write()`) u otros que devuelven el tipo o tamaño de la imagen (`type()`, `rows` o `cols`).

Tipos de dato:

El tipo de píxel (RGB, YUV, etc.) soportado por la clase `hls::Mat` es similar al utilizado en la biblioteca *OpenCV* original. En la figura 2.2 se observa los tipos soportados por dichas bibliotecas y como se nombran. Cogiendo como ejemplo `HLS_16UC3`, se define el tipo de píxel de tres canales con 16 bits sin signo por canal.

Tipos de píxel de hls::Mat					
Enteros	HLS_8UCX	HLS_16UCX	HLS_8SCX	HLS_16SCX	HLS_32SCX
Float	HLS_32FCX	HLS_64FCX			

Tipo de píxel de *OpenCV*					
Enteros	CV_8UCX	CV_16UCX	CV_8SCX	CV_16SCX	CV_32SCX
Float	CV_32FCX	CV_32FCX			

Significado de cada parámetro				
Párametros	**X**	**U**	**S**	**F**
Valores	[1, 4]	Entero sin signo	Entero con signo	Coma flotante

Tabla 2.2: Tipos de datos soportados

Para un correcto diseño, es importante elegir los tipos de datos adecuados para la síntesis del *core HW*. Un tipo más grande del necesario implicaría una mayor reserva de recursos de la plataforma y uno inferior introduciría errores de cálculo. Por ello, es importante hacer un estudio del tamaño y tipo de dato involucrado en el procesamiento. Para las plataformas FPGA el mejor tipo de dato a emplear es el entero, ya que ocupa muchos menos recursos y su procesamiento requiere de menos ciclos que el dato en coma flotante.

Interconexión de cores HW:

La conexión entre los filtros de la biblioteca *OpenCV* se realiza mediante la estructura típica *IplImage* o *cv::Mat* pero en los diseños acelerados por HW en *FPGA*s intervienen los siguientes dos tipos de conexiones:

- La primera es la conexión entre métodos de un mismo *core HW*. Se utiliza la estructura de datos *hls::Mat*.
- La segunda es la interconexión entre *cores HW*. La comunicación de entrada y salida del *core* se define mediante un bus de comunicación *AXI Stream*.

En la figura 2.4 se observan los métodos encargados de realizar la conversión entre el tipo *hls::Mat* y el tipo *AXI Stream* y viceversa. Gracias a

estos métodos se puede utilizar de forma trasparente el bus *AXI Stream* característico en la familia *Zynq*.

```
hls::AXIvideo2Mat <W, ROWS, COLS, TYPE> (
        hls::stream<ap_axiu<W,1,1,1>>& axi_video,
        hls::Mat<ROWS, COLS, TYPE>& img);

hls::Mat2AXIvideo <W, ROWS, COLS, TYPE> (
        hls::Mat<ROWS, COLS, TYPE>& img,
        hls::stream<ap_axiu<W,1,1,1>>& axi_video);
```

Figura 2.4: Métodos de conversión entre tipo *AXI Stream* y *hls::Mat*

Además, la biblioteca *HLS OpenCV* proporciona métodos para la conversión entre *cv::Mat* (estructura básica de *OpenCV*) y *Axi Stream* (véase figura 2.5). Dichos métodos son utilizados en los *testbench* para verificar el funcionamiento del *core HW* antes de proceder a su síntesis.

```
hls::cvMat2AXIvideo <W> (
        cv::Mat mat,
        hls::stream<ap_axiu<W,1,1,1>>& axi_video);

hls::AXIvideo2cvMat <W> (
        hls::stream<ap_axiu<W,1,1,1>>& axi_video,
        cv::Mat mat);
```

Figura 2.5: Métodos de conversión entre tipo *cv::Mat* y *AXi Stream*

Streams de datos:

Otra comunicación entre *cores HW* se basa en una comunicación punto a punto con un flujo continuo de datos (`streams`), es decir, con interfaces de entrada y salida del tipo *FIFO*. Debido a ello, la estructura de datos básica es diferente a la anterior y, además, no se utilizan las interfaces *AXI Stream*. Esta estructura está dividida en tres componentes:

- Un Flag de dato valido, usado en downstream.
- Un Flag de parada, usado en upstream.
- Tres componentes de datos: Coordenadas, Sincronías y Píxel.

En la figura 2.6 se describe la plantilla genérica con la cual se define cada estructura básica para cada caso de uso. Dicha plantilla contiene tres parámetros que la definen: el tipo de coordenadas, el tipo de dato de píxel y las sincronías.

```
template<typename PIXEL_T, typename SYNC_T, typename COORD_T>
struct VS_STREAM{
// Senyales de control
        VS_U1 HALT;
        VS_U1 VALID;
        VS_U1 ACTIVE;

// Componentes
        PIXEL_T PIXEL;
        SYNC_T SYNC;
        COORD_T COORD;
};
```

Figura 2.6: Streams de datos: Estructura Genérica

Tomando un ejemplo de caso de uso de esta estructura, éste estaría formado por un píxel del tipo *RBG* de 8 bits por canal, coordenadas x e y de 16 bits y cuatro flags para sincronías. En la figura 2.7 se observan las definiciones del ejemplo.

Píxeles	**Sincronías**	**Coordenadas**
Plantillas		

`template<int A,int D,int C>`
`struct vs_rgb{`
`  ap_uint<A> R;`
`  ap_uint<D> G;`
`  ap_uint<C> B;`
`};`
	-	`template<int A, int B>`
		`struct vs_coord{`
		`  ap_uint<A> X;`
		`  ap_uint<B> Y;`
		`};`

Definiciones		

`typedef vs_rgb<8,8,8>`
`VS_RGB8;`
	`typedef struct{`
	`  ap_uint<1> HSync;`
	`  ap_uint<1> VSync;`
	`  ap_uint<1> HBlank;`
	`  ap_uint<1> VBlank;`
	`} VS_SYNC;`
		`typedef vs_coord<16,16>`
		`VS_COOR16;`

Figura 2.7: Streams: Plantillas y ejemplo de píxel, sincronías y coordenadas

A partir de estas definiciones de tipo de píxel, coordenadas y sincronías del ejemplo se obtiene la definición de la estructura básica completa (véase figura 2.8).

```
typedef VS_STREAM<VS_RGB8, VS_SYNC, VS_COOR16> VS_STREAM_RGB;
```

Figura 2.8: Streams de datos: Ejemplo RGB 8 bits por canal

Además de estas estructuras, la biblioteca incluye una gran variedad de definiciones de tipos datos diseñados para reducir los recursos utilizados de la *FPGA*. En la figura 2.9 se observan los datos más típicos: entero con signo, entero sin signo, *linebuffers*, etc.

```
//Datos básicos
    typedef ap_int<16>  VS_S16;
    typedef ap_uint<16> VS_U16;
    typedef ap_int<32>  VS_S32;
    typedef ap_uint<32> VS_U32;

// Ventanas 3x3
    typedef ap_window<VS_S32,3,3>  WINDOWS;
    typedef ap_window<VS_RGB8,3,3> WINDOWS_RGB;

// Linebuffers para convoluciones
    typedef ap_linebuffer<VS_RGB8,2,MAX_WIDTH> RGB_BUF_2ROW;
    typedef ap_linebuffer<VS_RGB8,3,MAX_WIDTH> RGB_BUF_3ROW;
```

Figura 2.9: Definiciones de tamaño y tipo de datos básicos

Métodos y Operadores:

El siguiente punto describe los métodos y operadores incluidos en la biblioteca. Los filtros diseñados se encargan del procesado de los datos de entrada y los operadores son diseñados para realizar las operaciones aritméticas básicas que se deben soportar con los nuevos tipos de datos definidos. Estas funciones se intentan diseñar a partir de plantillas de *C++* siempre que sea posible.

En la figura 2.10 se presenta un ejemplo de un operador aritmético básico: la suma de dos datos del tipo *vs_rgb*.

```
template <int A, int D, int C>
vs_rgb<A,D,C> operator +(vs_rgb<A,D,C> E, vs_rgb<A,D,C> F)
   {
    vs_rgb<A,D,C> H;
    H.B = E.B + F.B;
    H.G = E.G + F.G;
    H.R = E.R + F.R;
    return H;
}
```

Figura 2.10: Operador suma de dos tipos vs_rgb

Los filtros diseñados tienen la misma funcionalidad que los filtros ori-

ginales de la biblioteca *OpenCV*. La diferencia más importante es definir qué parámetros van a ser fijos o variables cuando se sintetice el *core*. Tomando, como ejemplo, el método de conversión de un espacio de colores a otros de *OpenCV* (`cvtColor`) se generan los métodos equivalentes que son sintetizables tomando como valor fijo el espacio de color. Debido a esto, el **core** sintetizado solo transforma de un espacio fijo a otro fijo. En la figura 2.11 se muestra el ejemplo mencionado donde se clasifican los parámetros de cada método.

OpenCV	`void cvtColor(InputArray src, OutputArray dst, int code, int dstCn=0)`

	`Parámetros:`
	`src: Imagen de entrada.`
	`dst: Imagen de salida.`
	`code: Espacio de colores para conversión.`
	`dstCn: Número de canales de salida.`

hls	`template<typename CONVERSION,int SRC_T,int DST_T,int ROWS,int COLS>`
	`void CvtColor(`
	`hls::Mat<ROWS, COLS, SRC_T> &_src,`
	`hls::Mat<ROWS, COLS, DST_T> &_dst)`

	`Parámetros fijos en sintesis:`
	`SRC_T: Tipo de dato imagen de entrada.`
	`DST_T: Tipo de dato imagen de salida.`
	`CONVERSION: Espacio de colores para conversión.`
	`ROWS: Número de filas de la imagen.`
	`COLS: Número de columnas de la imagen.`
	`Parámetros variable:`
	`_src: Imagen de entrada.`
	`_dst: Imagen de salida.`

vs	`void vs_rgbtoy_filter_var(`
	`VS_STREAM_RGB In[MAX_HEIGHT][MAX_WIDTH],`
	`VS_S32 Out[MAX_HEIGHT][MAX_WIDTH], int rows, int cols)`

	`Parámetros fijos:`
	`MAX_HEIGHT: Tamaño máximo de filas.`
	`MAX_WIDTH: Tamaño máximo de columnas.`
	`Parámetros:`
	`In: Imagen de entrada.`
	`Out: Imagen de salida.`
	`rows: Número de filas de la imagen.`
	`cols: Número de columnas de la imagen.`

Figura 2.11: Conversor de espacios de colores de *OpenCV*, *HLS* y *Streams*

Verificación de los filtros:

Para verificar cada filtro de la biblioteca *HLS*, y no generar cada vez un sistema Hw/Sw completo para una plataforma *Zynq*, se diseña un proyecto puramente hardware, no se usa la parte del ARM. A diferencia de los anteriores diseños, éste se ha diseñado con la herramienta de *Vivado*. En la figura 2.12 se observa el funcionamiento de este diseño donde el flujo de datos es el siguiente:

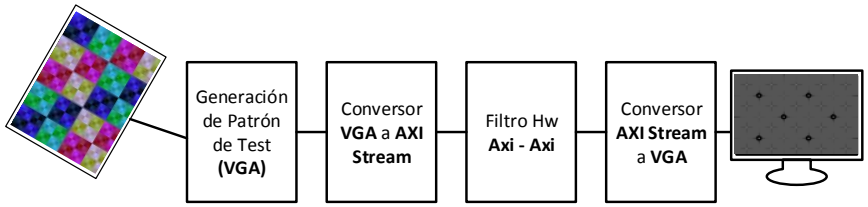


Figura 2.12: Configuración de test para los cores clonados

1 Se genera un patrón de test de vídeo para una salida con sincronismos Video Graphics Array (VGA)

2 Se transforma a una comunicación del tipo *AXI*.

3 Se filtra la imagen de entrada.

4 Se regeneran las sincronías VGA.

5 Se visualiza la imagen.

Con este sistema se puede verificar cada *core Hw* fácilmente.

2.1.2. Compilador AMAZynqC

El siguiente módulo es el compilador *AMAZynqC* que dirige el proceso de co-diseño. Su función se centra en obtener un modelo Hw/Sw soportado por las arquitecturas de referencia de *AMAZynq* que implemente el procesamiento de un sistema de visión definido en *C++* y *OpenCV*.

AMAZynqC es una modificación del *Clang*, el conocido compilador de C/C++ integrado en la infraestructura de compilación *LLVM*. Esta modificación extiende el *front-end* de compilación de *C++* de *Clang* añadiendo nuevos *pragmas* y parámetros de compilación para dar soporte a nuevas plataformas *SoC-FPGA* junto con sus bibliotecas Sw/Hw asociadas.

Estos *pragmas* permiten la especificación manual del particionado y el control de varios parámetros de bajo nivel como el ancho de palabra de los distintos flujos de datos. En la figura 2.13 se muestra un ejemplo de algunos de dichos *pragmas*.

Entrada	`#pragma amazynq_start:`
Salida	`#pragma amazynq_stop:`
Tipo de dato	`#pragma amazynq img.DT=TipoDato:`

Figura 2.13: `Pragmas` soportados por el compilador *AMAZynqC*

A partir de un proyecto Sw de entrada del compilador, éste tiene la misión de:

- Reconocer filtros nativos *OpenCV*.
- Reconocer nuevos filtros definidos mediante una concatenación de filtros nativos *OpenCV*.
- Recopilar información de alto nivel para la futura optimización de los filtros (linealidad, separabilidad, tipo y representación de los datos, dimensiones de los *kernels* de convolución,etc.).
- Reconocer los *pragmas* y directivas de compilación para controlar el proceso de síntesis de alto nivel.
- Soportar el acceso a una base de datos externa con información sobre la biblioteca predefinida de filtros Hw.
- Generar el elenco de ficheros descritos en la sección 2.2, que definen el proyecto de partida usando una plataforma de Hw reconfigurable (por el momento, *Zynq*).

La base de datos externa que se cita acomete un objetivo doble. Por un lado, en ella se lista los filtros *OpenCV* que son soportados por el compilador, y se permite al usuario decidir de forma controlada como se realizará la transformación a Hw reconfigurable. Por otro lado, permite la extensión del compilador con nuevos filtros definidos por el usuario. Por tanto, este archivo, define las reglas de transformación software a hardware.

La salida del compilador es un conjunto de tres proyectos nativos para las herramientas de *Xilinx* (*Vivado HLS* y *ISE*): *Proyecto HLS*, *Proyecto EDK* y *Proyecto Sw*.

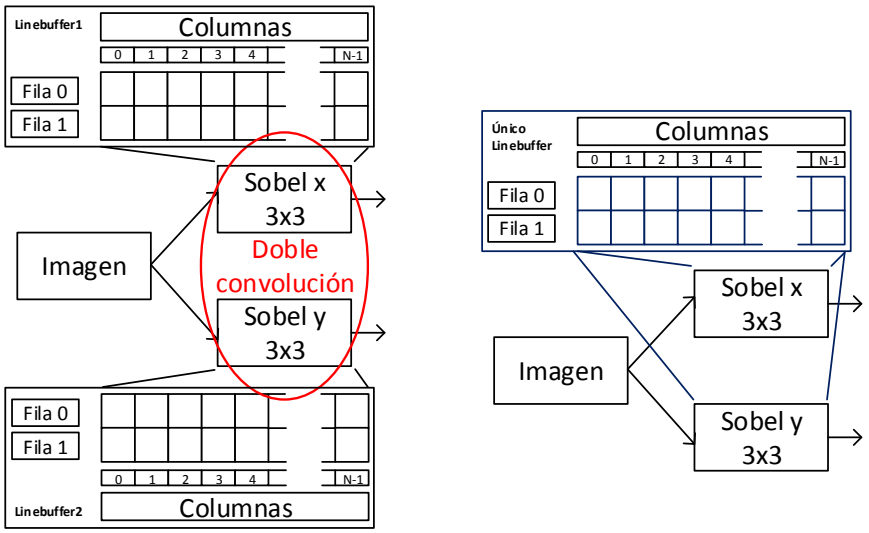
Optimizaciones

A parte de analizar todos los métodos, flujos o tipos de datos involucrados, el compilador debe realizar algunas optimizaciones para mejorar el rendimiento o reducir los recursos utilizados de la *FPGA*.

Estas optimizaciones se clasifican en dos tipos: **a nivel de arquitectura** o **a nivel de espacio de operador**.

Las optimización **a nivel de arquitectura** se basan en la utilización de los recursos empleados, es decir, se intenta compartir recursos entre las diferentes filtros involucrados. En el caso de una doble o triple convolución se puede compartir los *linebuffer* entre ellas cuando se cumplan ciertos requisitos.

En la figura 2.14 se presenta un ejemplo de una convolución doble (método *sobel*) y la optimización con el uso de un único *LineBuffer* para realizar el procesamiento.



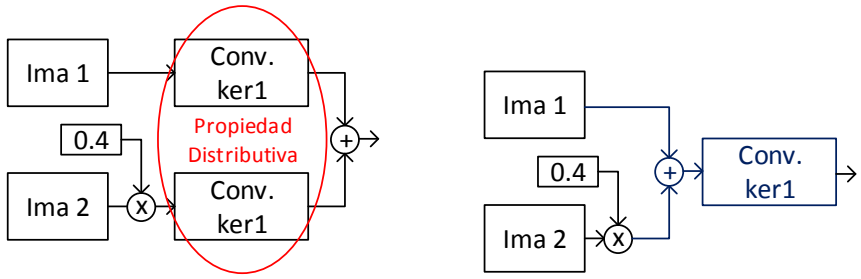
(a) Linebuffers independientes (b) Linebuffer único

Figura 2.14: Ejemplo de optimización a nivel de arquitectura

Este caso de optimización lo puede realizar *AMAZynqC* o bien la herramienta *Vivado HLS* de *Xilinx*.

El otro tipo de optimización, **nivel de espacio de operador**, intenta aprovechar las propiedades matemáticas de los operadores involucrados en el procesamiento, es decir, intenta reducir las operaciones involucradas a partir de sus propiedades. Por ejemplo, se puede reducir el número de convoluciones, siempre que cumplan unas especificas propiedades. Tomando un ejemplo de la propiedad distributiva se tiene $f * (g + h) = (f * g) + (f * h)$.

En la figura 2.15 se puede observar gráficamente el ejemplo $Ima1 * k1 + (Ima2 \cdot 0{,}4) * k1 = (Ima1 + Ima2 \cdot 0{,}4) * k1$.



(a) Sin aplicar propiedades (b) Uso de las propiedades

Figura 2.15: Ejemplo de optimización a nivel de espacio de operador

En el ejemplo anteriormente mencionado, se reduce el número de convoluciones. Por contra, se debe analizar el tamaño del tipo de datos para evitar que la nueva distribución del procesado no provoque errores en el cálculo (ejemplo a *overflow* u otros).

2.1.3. Arquitecturas de Referencia y Plantillas de Parametrización

Este módulo contiene las arquitecturas de referencia (*AR*) sobre las que *AMAZynqC* puede proyectar el diseño y una biblioteca de plantillas de parametrización que facilitan la especificación y reutilización de los núcleos de procesamiento Hw o Sw.

Arquitecturas de Referencia:

Una Arquitectura de Referencia se compone de todas las infraestructuras Hw y Sw necesarias que definen el sistema. La infraestructura Hw está formada por todas las instancias de los dispositivos involucrados, por ejemplo: configuración del *SoC-ARM* (relojes, Phase-Locked Loop (PLL), periféricos), buses de comunicación, controladores Acceso Directo a Memoria (DMA), núcleos IP, etc. Y la infraestructura Sw está formada por las bibliotecas, drivers, archivos de soporte del Sistema Operativo, etc.

AMAZynq incluye una base de datos de arquitecturas de referencia que soportada el compilador. Según las especificaciones o la exigencia del sistema, el *core Hw* generado por la herramienta *Vivado HLS*, será instanciado en una arquitectura u otra.

Hoy en día, *Xilinx*, *Analog Devices* y otras compañías proporcionan varios diseños [18, 19] para la plataforma *Zynq* que se pueden utilizar como arquitectura de referencia. Además, esta base de datos está preparada para ser enriquecida con las aportaciones de terceras personas.

Hay muchos casos de uso en sistemas de visión y, por ello, es necesario muchas arquitecturas de referencia. En la figura 2.16 se muestra cuatro posibles arquitecturas.

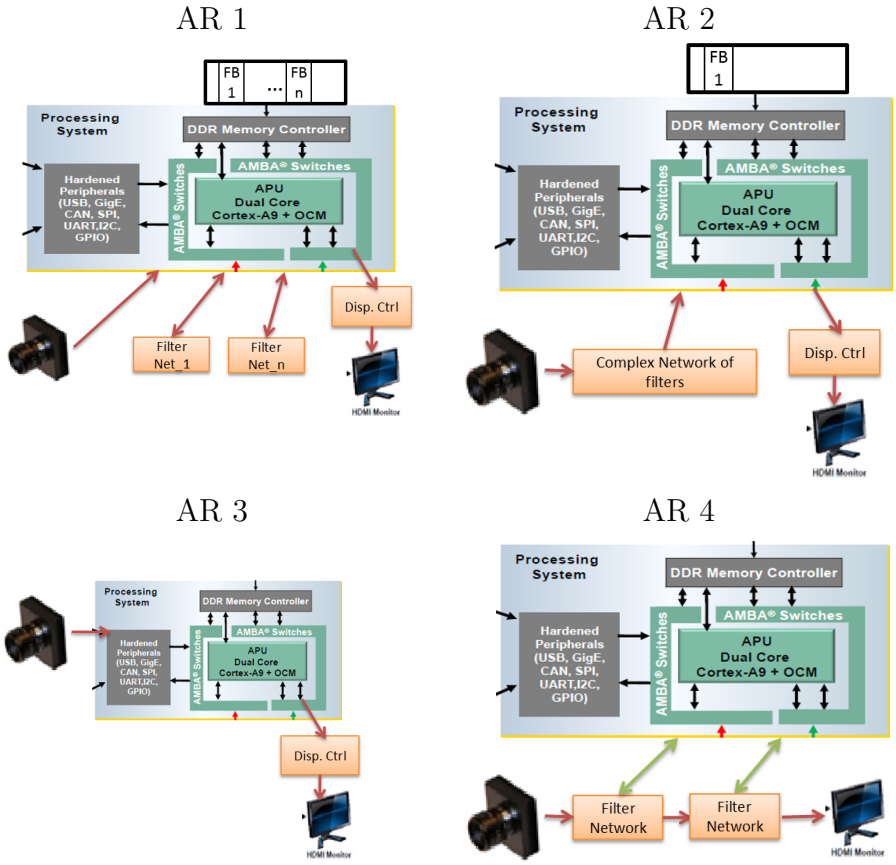


Figura 2.16: Ejemplos de Arquitecturas de Referencia

La diferencia más importante entres las arquitecturas anteriores mencionadas, se centra en el modo de comunicación que tiene el *ARM* con los *cores Hw*. El funcionamiento de cada arquitectura se basa en:

- La arquitectura *AR1* tiene como característica principal una comunicación fija mediante un bus para cada filtro, cámara y monitor. Es decir, cada módulo está conectado y controlado únicamente por la CPU. Debido a esto se necesitan tantos *framebuffers* como filtros.

- La arquitectura *AR2* posee un sistema más sencillo que el anterior. La cámara se conecta directamente a una red de filtros interconectados, la red se conecta a la CPU y el monitor a la CPU. En este caso sólo hace falta un *framebuffer*.
- La arquitectura *AR3* es el sistema más sencillo y no contiene ningún filtro Hw de procesado de imagen y todo el procesado lo realiza la CPU.
- En la arquitectura *AR4*, la CPU sólo tiene el control de los filtros. La cámara se conecta al filtro1, el filtro1 al filtro2 y del filtro2 al monitor. En este caso no hace falta ningún *framebuffer*. Por el contrarío, no se puede almacenar la imagen.

Los *framebuffers* son módulos que gestionan una reserva de memoria donde se almacenan las imágenes. En estos ejemplos, cada framebuffer suele tener un tamaño de cuatro imágenes *full-hd* de 32 bits por píxel.

Plantillas de Parametrización:

Las *plantillas* son código invariante que ayuda al compilador a generar los proyectos de salida. Estas definen parte del código de los *cores Hw*, parte de las aplicaciones software (para ARM) y los *testbench* que se pueden ejecutar en la herramienta *Vivado HLS*.

Entre las estructuras constantes destacan tres partes de código:

Las directivas o *pragmas* para las interfaces de entrada y salida de los *core Hw*. Las interfaces son del tipo *First In First Out (FIFO)* debido a que las comunicaciones por bus son del tipo *AXI-Stream*. Como se observa en las declaraciones, el bus posee cinco señales, una de datos y cuatro de control (ver figura 2.17).

```
#pragma HLS RESOURCE variable=inter_pix core=AXI4Stream
   metadata="-bus_bundle␣INPUT_STREAM"
#pragma HLS RESOURCE variable=inter_pix core=AXIS port_map =
   { {inter_pix_data_V TDATA} {inter_pix_user_V TUSER} {
   inter_pix_last_V TLAST} {inter_pix_tdest_V TDEST} {
   inter_pix_strb_V TSTRB} }
#pragma HLS INTERFACE ap_fifo port=inter_pix
```

Figura 2.17: Interfaz de entrada y salida del *core HW*

Las declaraciones de conectores entre las uniones entre los distintos filtros. Para conectar una salida de un filtro con otro, se debe instanciar una variable

auxiliar de tipo FIFO, distinguiendo la parte de síntesis y la de simulación (ver figura 2.18).

```
#ifndef __SYNTHESIS__
VS_S32 (*in_vs)[MAX_HEIGHT][MAX_WIDTH] = (VS_S32(*)[
   MAX_HEIGHT][ MAX_WIDTH]) malloc (((MAX_HEIGHT)*( MAX_WIDTH
   ))* sizeof (VS_S32));
#else
VS_S32 _in_vs [MAX_HEIGHT][MAX_WIDTH];
VS_S32 (*in_vs)[MAX_HEIGHT][MAX_WIDTH] = &_in_vs;
#endif
```

Figura 2.18: Variables definidas para simulación y síntesis

La estructura de test. El código de test es siempre el mismo proceso que se encarga de realizar lectura, procesado y escritura de una imagen. Para ello, lo único que modifica es la instancia al nuevo Hw que suele ser un par líneas.

2.2. Flujo de diseño

El siguiente bloque describe el flujo de diseño de *AMAZynq*. En la figura 2.19 se observa dicho flujo, a partir de un código fuente con *pragmas*, *AMAZynqC* identifica cada parámetro de cada función de *OpenCV* y los clasifica por su naturaleza entre: flujo de vídeo, parámetro en tiempo de compilación (**ct**) y parámetro en tiempo de ejecución (**rt**).

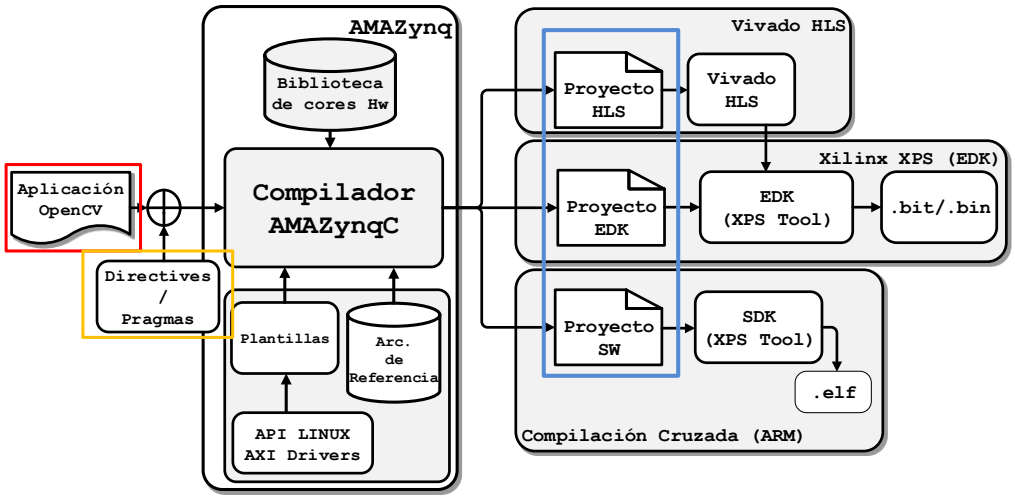


Figura 2.19: Flujo de Diseño y Diagrama de Bloques de AMAzynq

Los parámetros **ct** son constantes y se utilizan para guiar la síntesis de las primitivas *OpenCV*, por lo tanto, no es posible su modificación durante el tiempo de ejecución del sistema. Los **rt**, por el contrario, se pueden utilizar para modificar los procesos llevados a cabo por el *core Hw*. Esta información es de suma importancia para que *AMAZynqC* pueda sintetizar adecuadamente el sistema.

La importancia que un parámetro sea **ct** o **rt** es debido a la estructura que se genera en el *core Hw*. Un parámetro **ct** implica una generación fija e invariante en el core y un parámetro **rt** implica la generación de una variable con sus interfaces de comunicación (ejemplo *AXI*).

2.2.1. Análisis del código de entrada

La primera misión que tiene *AMAZynqC* es analizar el archivo de entrada, detectar la región a acelerar por Hw (definida entre *pragmas*) y obtener la información de las funciones involucrados y los flujos entre ellas. En la figura 2.20 se pone un ejemplo para analizar por el compilador.

```
int dx = 1, dy = 1;
int ksize = 3, scale = 1, delta = 0;

#pragma amazynq_start:
cv::cvtColor(src, src_gray, CV_BGR2GRAY);
cv::Sobel(src_gray, dst, ddepth, dx, dy, ksize, scale, delta,
    DORDER_DEFAULT);
#pragma amazynq_stop:
```

Figura 2.20: Instancias de los métodos de OpenCV

En este ejemplo, se presenta dos métodos sencillos de la biblioteca *OpenCV*. El primer método es un conversor de espacio de colores (cv::cvtcolor) y el segundo es una convolución para detección de bordes (cv::sobel).

En este punto, el compilador comprueba si los métodos se soportan para síntesis. Cuando se comprueba que son sintetizables, *AMAZynqC* analiza las cabecera de cada método (véase figura 2.21) y realiza un análisis de cada parámetro de las instancias.

La información obtenida tiene dos partes: la primera información es la del método (nombre, retorno, número de parámetros) y la segunda información es la de cada parámetro.

```
void cvtColor(
        InputArray src, OutputArray dst,
        int code, int dstCn=0)

void Sobel(
        InputArray src, OutputArray dst,
        int ddepth, int dx, int dy,
        int ksize=3, double scale=1, double delta=0,
        int borderType=BORDER_DEFAULT)
```

Figura 2.21: Cabecera de los métodos OpenCV

En la figura 2.22 se observa la información sacada para la instancia *cv::cvtColor*. La información más importante que se saca del primer método

son el número de parámetros (son 3), que no tiene retorno la función y que tiene un valor constante en el tercer parámetro (`CV_BGR2GRAY`).

Instancia 1: cv::cvtColor(src, src_gray, CV_BGR2GRAY			
Método	Nombre	cv::cvtColor	
	Return	None	
	Parámetros	3	
			Función Original
Param 1	Nombre	src	src
	Tipo	cv::Mat	InputArray
Param 2	Nombre	src_gray	dst
	Tipo	cv::Mat	OutputArray
Param 3	Nombre	CV_BGR2GRAY	code
	Tipo	constante	int

Figura 2.22: Información de las instancias del cv::cvtcolor

Del segundo método (figura 2.23) se obtiene una información similar a la anterior. El método *cv::sobel* tiene 9 parámetros de los cuales todos menos los métodos de las imágenes de entrada y salida son constantes.

Instancia 2: cv::Sobel(src_gray, dst, ddepth, dx, dy, ...)			
Método	Nombre	cv::Sobel	
	Return	None	
	Parámetros	9	
			Función Original
Param 1	Nombre	src_gray	src
	Tipo	cv::Mat	InputArray
Param 2	Nombre	dst	dst
	Tipo	cv::Mat	OutputArray
Param 3	Nombre	ddepth	ddepth
	Tipo	int	int
	Value	1	
...			
Param 9	Nombre	-	borderType
	Tipo	int	int

Figura 2.23: Información de las instancias del cv::sobel

En este punto, se debe entender que implica que un parámetro de entrada sea constante o sea una variable. Como se menciona al principio del apartado 2.2.1, que un parámetro sea constante implicará una parte del **core HW** fijo e invariante. Pero, si el parámetro es variable, genera una variable y unos interfaces de comunicación (*AXI Slave*) en el core Hw.

A partir de aquí, el compilador busca en la biblioteca de clones Hw (*Biblioteca HLS*) los métodos clones de los originales. En la figura 2.24 se observa los prototipos a esos métodos. En dicha figura se observa los parámetros que forman el prototipo y en algunas ocasiones el método clonado tiene menos parámetros que en el método original. A pesar de esto, el compilador obtiene una correcta relación entre cada parámetro del método original y los parámetros del método clonado. Para ello, comprueba si los nombres de cada parámetro son iguales e ignora los que no son necesarios.

```
template<typename CONVERSION, int SRC_T, int DST_T, int ROWS,
    int COLS>
void CvtColor(
        Mat<ROWS, COLS, SRC_T> &_src,
        Mat<ROWS, COLS, DST_T> &_dst)

template<int XORDER, int YORDER, int SIZE, int SRC_T, int
   DST_T, int ROWS, int COLS, int DROWS, int DCOLS>
void Sobel (
        Mat<ROWS, COLS, SRC_T> &_src,
        Mat<DROWS, DCOLS, DST_T> &_dst)
```

Figura 2.24: Cabecera de los métodos hls::OpenCV

En el último punto, el compilador genera las instancias de los métodos clonados, declaración de variables, interfaces de comunicación, etc. y genera el proyecto que es utilizado para sintetizar el *core Hw*.

```
hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC3> src(rows, cols);
hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_8UC1> src_gray(rows, cols
   );
hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_16SC1> dst(rows, cols);

int dx = 1, dy = 1;
int ksize = 3, scale = 1, delta = 0;

hls::CvtColor <HLS_RGB2GRAY, HLS_8UC3, HLS_8UC1, MAX_HEIGHT,
   MAX_WIDTH> (src, src_gray);
hls::Sobel <dx, dy, ksize, HLS_8UC1, HLS_16SC1, MAX_HEIGHT,
   MAX_WIDTH, MAX_HEIGHT, MAX_WIDTH> (src_gray, dst);
```

Figura 2.25: Instancias de los métodos hls::OpenCV

Por último, en dichas instancias se encuentran unos parámetros que son *MAX_HEIGHT* y *MAX_WIDTH* los cuales no aparecen en el código original. Estos parámetros son necesarios ya que en los diseños Sw no hace falta indicar el tamaño de la imagen, esta información se gestiona en tiempo de ejecución

pero en las FPGA se tiene que configurar de antemano. Esta información es importante para la generación de los *linebuffers*.

2.2.2. Proyectos de salida de AMAZynq

A partir de este análisis, el compilador genera tres proyectos de salida: *Proyecto HLS*, *Proyecto EDK* y *Proyecto Sw*.

Síntesis del Hw: Proyecto HLS

El *Proyecto HLS* incluye la descripción para su síntesis de alto nivel (*HLS*) de un *core Hw*, la especificación de la interfaz con los correspondientes buses *AXI* y un banco de pruebas. *AMAZynqC* extrae las funciones y dependencias de los datos desde el código original y construye el *core Hw* equivalente utilizando las primitivas *HLS OpenCV*. Además de esta tarea, es posible llevar a cabo optimizaciones a diferentes niveles (nivel de espacio de operadores, nivel de arquitecturas o nivel de representación de los datos). El banco de pruebas permite realizar de forma automática la verificación funcional del *core* y la exploración automática de su espacio de diseño.

Además, con el uso de *Vivado HLS*, se puede lanzar una simulación automáticamente para verificar la funcionalidad del *core* frente el código *OpenCV* original, a partir de imágenes fijas, archivos de vídeo o vídeo en directo (cámara web) (ejemplo véase en figura 2.26). La simulación realizada puede revelar diferencias en la exactitud, lo que puede indicar que se debe ajustar la anchura de los datos con *AMAZynqC*.

Una vez verificado, el *core* es sintetizado por las herramientas de *Vivado HLS*. Como resultado se produce un core IP que se integrará en el *Proyecto EDK* por medio de adaptadores e interfaces de bus apropiados. Estas interfaces se declaran y se configuran automáticamente por *AMAZynqC* automáticamente.

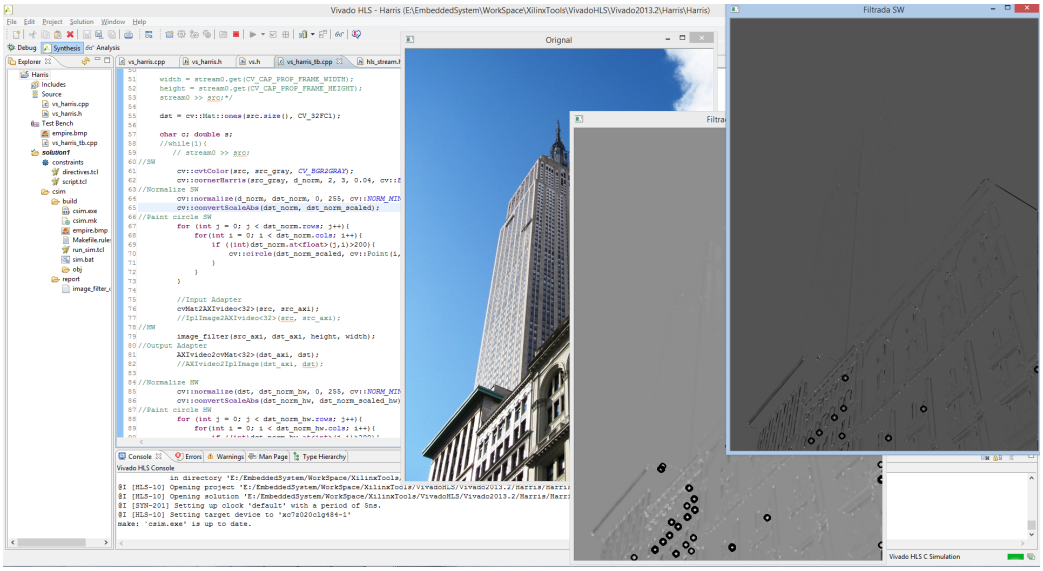


Figura 2.26: Proyecto HLS: Ejecución del test en *Vivado HLS*

Definición de la Arquitectura: Proyecto EDK

El *Proyecto EDK* se define por medio de un archivo *Xilinx Hardware Specification file* (`.mhs`). Este archivo hace uso de una arquitectura de referencia específica, que incluye la instancia del *core Hw* sintetizado por *Vivado HLS* y sus infraestructuras de apoyo, como *VideoDMA*, *AXI buses*, etc.

El proyecto se sintetiza mediante el uso de la herramienta *EDK* de *Xilinx* y se obtiene el archivo de configuración para la plataforma *Zynq*. En la figura 2.27 se puede observar una ejecución del entorno *EDK*.

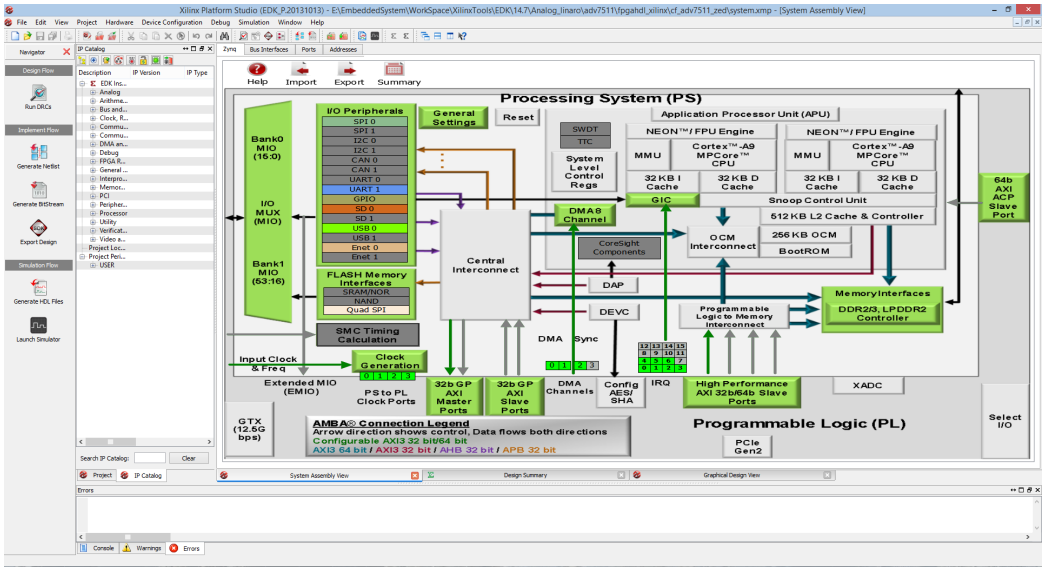


Figura 2.27: Proyecto EDK: Herramienta EDK de Xilinx

Aplicación de usuario: Proyecto Sw

El *Proyecto Sw* comprende la parte Sw del *código OpenCV* original y las llamadas a métodos para el manejo de las comunicaciones entre la parte Sw y Hw. Este Sw es una aplicación *Linux* para *ARM*.

El proyecto se compila mediante la herramienta *SDK* de *Xilinx*, véase figura 2.28.

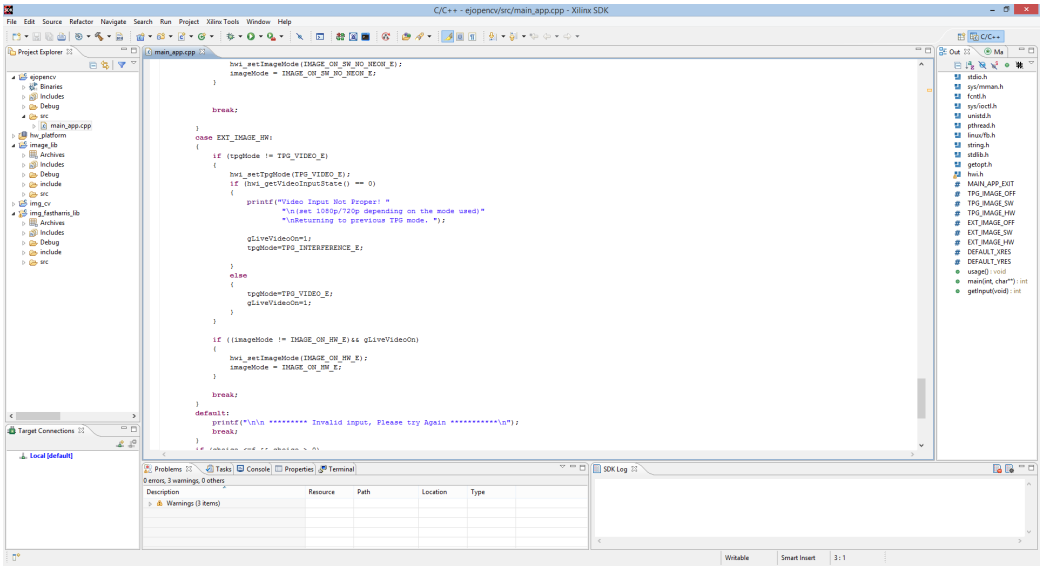


Figura 2.28: Proyecto SW: Entorno SDK para Cross-compilación

Capítulo 3

Casos de Estudios

Con el fin de evaluar el entorno *AMAZynq*, se ha definido un caso de estudio sencillo donde se comprueba su funcionamiento y se exploran distintas particiones en el espacio de co-diseño de un sistema de procesamiento de vídeo. El código fuente de este caso de estudio ha sido tomado de un ejemplo de la documentación online de *OpenCV* [20].

Particularmente, dicho ejemplo implementa un caso de uso del algoritmo de detección de esquinas *Harris Corner Detector* [21] (véase figura 3.1). Este algoritmo esta formado de unos pesos (ω_k), y unas matrices de que corresponden al gradiente en x (I_x) y en y (I_x) de la imagen de entrada. Se trata de un algoritmo de complejidad media, bien estudiado e implementado en diferentes plataformas [22, 23].

$$G(x,y) = \begin{pmatrix} \sum\limits_{x_y,y_k \epsilon W} \omega_k I_x^2 & \sum\limits_{x_y,y_k \epsilon W} \omega_k I_x I_y \\ \sum\limits_{x_y,y_k \epsilon W} \omega_k I_x I_y & \sum\limits_{x_y,y_k \epsilon W} \omega_k I_y^2 \end{pmatrix} \tag{3.1}$$

$$c(x,y) = Det(G(x,y)) - k \cdot trace^2(G(x,y)) \tag{3.2}$$

Figura 3.1: Algoritmo del detector del Harris Corner

En la figura 3.2 se observan los bloques y el flujo de diseño que forma el detector *Harris Corner* para el caso de uso descrito en la documentación mencionada de *OpenCV*.

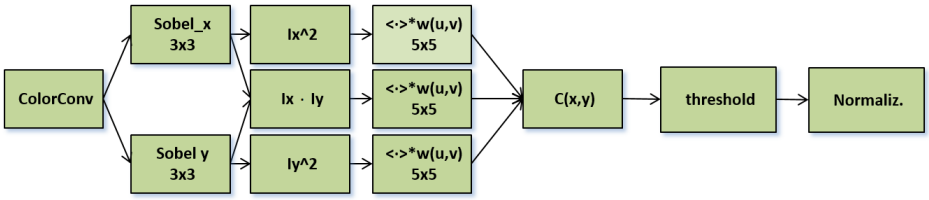


Figura 3.2: Flujo de datos del Detector Harris Corner

Los bloques más importantes del diseño son cinco convoluciones (dos sobel y tres con un kernel $5x5$). El procesado empieza con una imagen de entrada que es convertida a escala de grises y termina la misma imagen de entrada (en color) con una superposición de pequeños círculos en los lugares donde se han detectado esquinas.

Para la implementación del sistema, se destaca que la única modificación añadida al código fuente es la adición de los *pragmas* (fig. 3.3) soportados por *AMAZynq*. Por lo tanto, y como novedad a señalar en esta metodología, el código modificado (código C++ que usa la biblioteca OpenCV), es directamente compatible con los compiladores *GCC* y *Clang* sobre arquitecturas como *Intel32*, *AMD64* o *ARM*.

```
                    //Pragma Entrada
#pragma amazynq_start:

                    //Pragma Salida
#pragma amazynq_stop:
```

Figura 3.3: *Pragmas* que definen la región a acelerar por HW

Para realizar la exploración del espacio de diseño, se han utilizado los métodos internos de la función principal `cv::cornerHarris(...)` (véase figura 3.4) incluida en la biblioteca *OpenCV* y, así, obtener la solución óptima del sistema. El estudio también realiza distintas exploraciones de los tipos y tamaños de datos con el fin de reducir el área del diseño ocupada por la *FPGA*.

```
...
#pragma amazynq_start:          //Pragma Entrada
cvtColor (src, gray, CV_BGR2GRAY);
Sobel(gray, dx, depth, 1, 0, 3, scale, 0, BORDER_DEFAULT);
Sobel(gray, dy, depth, 0, 1, 3, scale, 0, BORDER_DEFAULT);
...
multiply(dx, dx, dxx, 1, muldepth);
multiply(dy, dy, dyy, 1, muldepth);
multiply(dx, dy, dxy, 1, muldepth);
...
boxFilter(dxx, Fx, Boxdepth, Size(5, 5), anchor, 0,
    BORDER_DEFAULT);
boxFilter(dyy, Fy, Boxdepth, Size(5, 5), anchor, 0,
    BORDER_DEFAULT);
boxFilter(dxy, Fxy, Boxdepth, Size(5, 5), anchor, 0,
    BORDER_DEFAULT);
...
calcHarris(Fx, Fy, Fxy, dst, 0.04);
#pragma amazynq_stop:           //Pragma Salida
...
```

Figura 3.4: Ejemplo del uso de los *pragmas* de *AMAZynq* para acelerar el filtro *Corner Harris*.

3.1. Caso 1: Diseño de un sistema acelerado

El primer caso de estudio se basa en verificar un sistema completo funcional en la plataforma *Zedboard*. Dicha plataforma se ha configurado siguiendo el esquema de conexión de periféricos mostrados en la figura 3.5. Consta de una cámara conectada por *usb* al *ARM* para la captación de las imágenes a procesar y un monitor conectado por *HDMI* para la visualización.

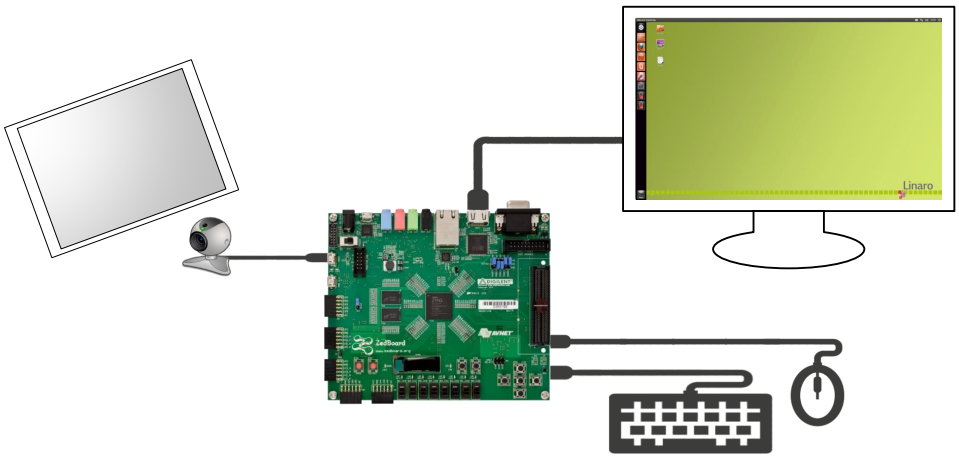


Figura 3.5: Esquema de conexión de *Zedboard* para el caso de estudio Harris Corner

Para generar el sistema completo, el compilador *AMAZynqC* obtiene el flujo de datos del código original Sw (partición llamada *ALL_SW*) donde se obtienen las relaciones de todos los tipos de datos y conexiones involucradas.

Además, *AMAZynqC* genera una tabla con los tipos de datos soportados por cada método involucrado y un grafo dirigido del flujo (descrito en *Graphviz-DOT*) del sistema de entrada. En la figura 3.6 se observa los tipos de datos y el flujo del programa, donde se empieza reservando memoria suficiente para todas las imágenes involucradas (entrada, intermedias y salida) y, a continuación, entra en un ciclo de captura, procesado y visualización.

Type	In	Out
Cam	—	CV_8UC3
cvtColor	CV_8UC3	CV_8UC1
sobel	CV_8U	CV_16S
		CV_32F
BoxFilter	CV_32S	CV_32S
	CV_32F	CV_32F
calcHarris	CV_32S	CV_32F
	CV_32F	CV_32F

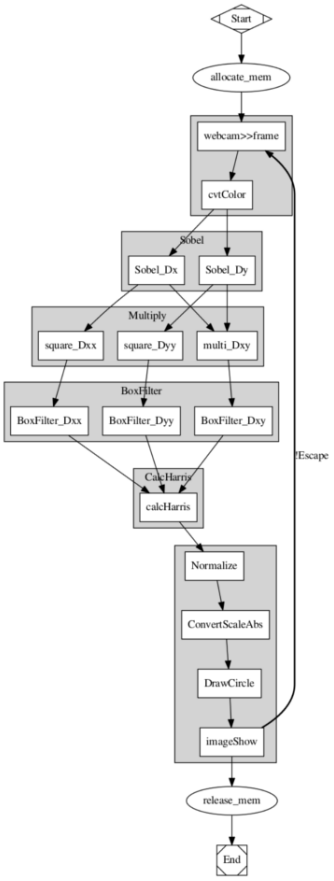


Figura 3.6: Tipos y Flujo de datos del Harris Corner (ALL_SW)

A continuación se describe cada paso del bucle antes mencionado.

1 Se captura una imagen de la cámara (RGB de 8 bits por canal).

2 Se convierte la imagen RGB a escala de grises con 8 bits de color (`cv::cvtColor()`).

3 Se realiza el gradiente de la intensidad en las direcciones x e y de la imagen (`cv::sobel()`) obteniendose un tamaño de 16 bits por píxel (entero con signo) o 32bits por píxel en coma flotante. Se emplea un *kernel* de dimensión 3×3.

4 Se multiplica píxel a píxel las matrices resultantes obteniendose a la salida un entero con signo de 32 bits por píxel o coma flotante de 32 bits.

5 Se realiza un emborronado de las imágenes (`cv::BoxFilter()`) donde se emplea un *kernel* de dimensión 5×5. El píxel de salida es del mismo tipo que el de entrada (entero con signo de 32 bits o coma flotante de 32 bits).

6 Se etiquetan las esquinas usando la función `cv::calcHarris()`. Los píxeles de salida son del tipo coma flotante de 32 bits.

7 Se normalizan los valores en el rango de enteros de 0 a 255, se dibujan las esquinas detectadas y se visualizan.

Una vez definido el funcionamiento del proyecto Sw, se define la parte que es acelerada por Hw. En la figura 3.7 se aprecia los *pragmas* con el código original. Por tanto, los *pragmas* (`#pragma amazynq_start/stop`) controlan la ejecución en una arquitectura de un microprocesador o en un *core* diseñado a partir del código *C++* encerrado.

```
...
#pragma amazynq_start:          //Pragma Entrada
cvtColor (src, src_gray, CV_BGR2GRAY);
...
calcHarris(Fx, Fy, Fxy, dst, 0.04);
#pragma amazynq_stop:           //Pragma Salida
...
```

Figura 3.7: Código original con *pragmas* que definen la partición Hw/Sw

3.1.1. Proyectos de salida de Harris Corner

Como se menciona en el Apartado 2.2.2 el compilador *AMAZynqC* genera tres proyectos de salida. Cada uno tiene una función especifica para la generación del sistema completo.

Primera salida: Proyecto HLS

El primer proyecto que se genera es el *Proyecto HLS* que contiene unos *scripts* para síntesis soportados por la herramienta *Vivado HLS*. *AMAZynqC* ejecuta estos *scripts* y obtiene el *core Hw* sintetizado, el cual se encarga de acelerar parte del algoritmo de *Harris*.

En la figura 3.8 se observa los métodos que define el **core Hw** que son clonados de los originales de *OpenCV*. Dichos métodos difieren de los originales en que definen de antemano el tipo y tamaño de las imágenes, como se observa en los parámetros de la plantilla: `MAX_HEIGHT, MAX_WIDTH` y `HLS_32SC1`.

```
...
hls::Mat<MAX_HEIGHT,MAX_WIDTH,graydepth> gray(rows,cols);
hls::CvtColor<HLS_RGB2GRAY, HLS_8UC3, HLS_8UC1, MAX_HEIGHT,
    MAX_WIDTH>(_src, gray);
...
//Primer Sobel
hls::Mat<MAX_HEIGHT, MAX_WIDTH, sobeldepth> dx(rows, cols);
hls::Sobel<1, 0, 3, HLS_8UC1, HLS_32SC1, MAX_HEIGHT,
    MAX_WIDTH, MAX_HEIGHT, MAX_WIDTH>(gray1, dx);
hls::Mat<MAX_HEIGHT, MAX_WIDTH, HLS_32SC1> dy(rows, cols);
//Segundo Sobel
hls::Sobel<0, 1, 3, HLS_8UC1, HLS_32SC1, MAX_HEIGHT,
    MAX_WIDTH, MAX_HEIGHT, MAX_WIDTH>(gray2, dy);
...
hls::calcHarris<HLS_32SC1, HLS_32SC1, HLS_32SC1, HLS_32SC1,
    HLS_32FC1, MAX_HEIGHT, MAX_WIDTH>(dxx_p, dxy_p, dyy_p,
    _dst, 0.04);
...
```

Figura 3.8: Métodos clonados del *Detector Harris* en el *core Hw*

A parte de los métodos clonados, hace falta adaptadores de señal. En la figura 3.9 se observa los adaptadores de entrada y salida que convierten la señal del tipo *AXI Stream* al tipo adecuado que necesita el *core*, en este caso de estudio *hls::Mat*

```
// Input Adapter
hls::AXIvideo2Mat<32, MAX_HEIGHT, MAX_WIDTH, HLS_8UC3>(
    inter_pix, _src);
...
// Output Adapter
hls::Mat2AXIvideo<32, MAX_HEIGHT, MAX_WIDTH, HLS_32FC1>(_dst,
     out_pix);
```

Figura 3.9: Adaptadores de entrada entre *hls::Mat* y *AXI*

Además, se necesita definir las interfaces de entrada y salida del *core Hw*. En la figura 3.10 se observa dichas directivas mínimas necesarias que indican al sintetizador de *Vivado HLS* el tipo de comunicación que debe instanciar.

```
...
#pragma HLS RESOURCE variable=inter_pix core=AXIS metadata="-
    bus_bundle␣INPUT_STREAM"
//OUT
#pragma HLS RESOURCE variable=out_pix core=AXIS metadata="-
    bus_bundle␣OUTPUT_STREAM"
//CONTROL
#pragma HLS RESOURCE core=AXI_SLAVE variable=return metadata=
    "-bus_bundle␣CONTROL_BUS"
...
```

Figura 3.10: Directivas de los interfaces del *core Hw*

Por otra parte, el compilador genera un grafo descrito en lenguaje *DOT* del flujo de datos del *core Hw* (véase figura 3.11). Dicho flujo difiere del Sw en los adaptadores de entrada y salida, en las *FIFOs* y en los tipos de datos.

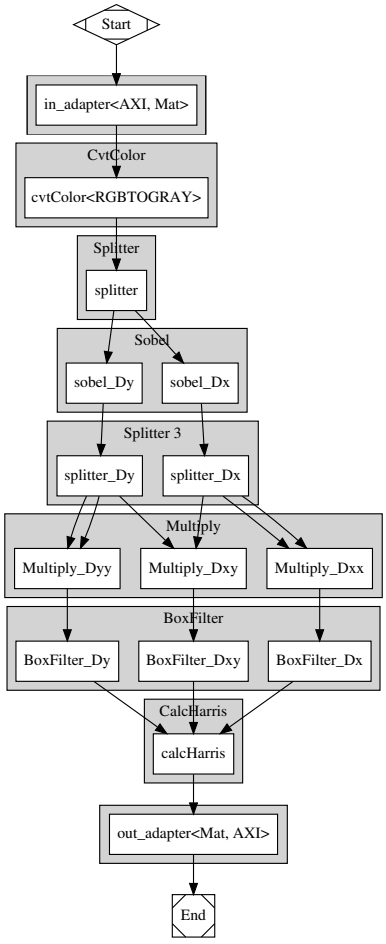


Figura 3.11: Flujo de datos del *core Hw* (*ALL_HW*)

A continuación se describe las diferencias entre el flujo de datos Hw y el original:

- Se adapta la señal de entrada del tipo *AXI* a tipo *hls::mat()*.
- Se clona la salida del *cvtColor* (*splitter*) para obtener dos copias a la salida.
- Se clona la salida de cada *sobel* (*splitter*) para obtener tres copias por cada entrada.
- Se adapta la señal resultante del tipo *hls::mat()* al tipo *AXI*.

Para el correcto funcionamiento del núcleo, se incluye unos *splitters* para realizar múltiples lecturas en paralelo de un valor. Estos *splitters* son necesarios en las entradas del método *sobel* y *Multiply*.

Por último, en la tabla 3.1 se muestra la elección del tipo de dato a utilizar en cada método involucrado en el *core HW*.

Type	**In**	**Out**
In Adapter	`32 bits`	`HLS_8UC3`
hls::cvtColor	`HLS_8UC3`	`CV_8UC1`
hls::sobel	`HLS_8UC1`	`CV_16SC1`
Multiply	`HLS_16SC1`	`CV_32SC1`
hls::BoxFilter	`HLS_32SC1`	`CV_32SC1`
hls::calcHarris	`HLS_32SC1`	`CV_32FC1`
Out Adapter	`HLS_32FC1`	`32 bits`

Tabla 3.1: Tipos de Datos soportados en el *core Hw*

A continuación se describe el tipo de dato de entrada y salida de cada método.

- `cvtColor`: Entrada RGB de 8bits por canal y salida 8bits sin signo.
- `Sobel`: Entrada 8bits sin signo y salida entero 16bits con signo.
- `Multiply`: Entrada entero 16bits con signo y salida entero 32bits con signo.
- `BoxFilter`: Entrada entero 32bits con signo y salida entero 32bits con signo.
- `calcHarris`: Entrada entero 32bits con signo y salida en coma flotante de 32 bits.

Además, el *proyecto HLS* incluye un *testbench* que se puede ejecutar en la herramienta *Vivado HLS* y comprobar el funcionamiento del *core HW* (figura 3.12). En este test se verifica la funcionalidad del algoritmo del *core Hw* pero no se puede comprobar el funcionamiento de los interfaces.

(a) Imagen de entrada (b) Procesado SW (c) Procesado HW

Figura 3.12: Test Funcional realizado con la herramienta *Vivado HLS*

Segunda Salida: Proyecto EDK

El segundo proyecto que se genera es el *Proyecto EDK*. Este proyecto incluye todas las infraestructuras Hw del sistema que se definen en la arquitectura de referencia (véase fig. 3.14) y, además, la instancia del *core HW* generado en el *Proyecto HLS*. Este proyecto está formado por unos *scripts* compatibles con la herramienta de *EDK* de *Xilinx* encargada de implementar el sistema, y un archivo de extensión `.mhs` (utilizado por la herramienta *EDK*), que contiene todas las instancias de los *cores*. En la figura 3.13 se observa una parte de las instancias.

```
BEGIN processing_system7
    PARAMETER INSTANCE = processing_system7_0
    PARAMETER HW_VER = 4.01.a
    PARAMETER C_DDR_RAM_HIGHADDR = 0x1FFFFFFF
    ...
END
BEGIN hardware_filter_top
    PARAMETER INSTANCE = FILTER_ENGINE
    PARAMETER HW_VER = 1.02.a
    PARAMETER C_S_AXI_CONTROL_BUS_BASEADDR = 0x400d0000
    PARAMETER C_S_AXI_CONTROL_BUS_HIGHADDR = 0x400dffff
    BUS_INTERFACE S_AXI_CONTROL_BUS = axi4_lite
    BUS_INTERFACE OUTPUT_STREAM = FILTER_DMA_S2MM
    BUS_INTERFACE INPUT_STREAM = FILTER_DMA_MM2S
    ...
END
```

Figura 3.13: Archivo system.mhs: Instancias de cores

Para su implementación, *AMAZynqC* hace uso de los *scripts* y obtiene el archivo de configuración de la *FPGA* (*system.bit*). Para la generación del archivo de arranque para la plataforma con *Zynq* (*boot.bin*), es necesario generarlo manualmente con la herramienta *bootmanager* del *SDK*.

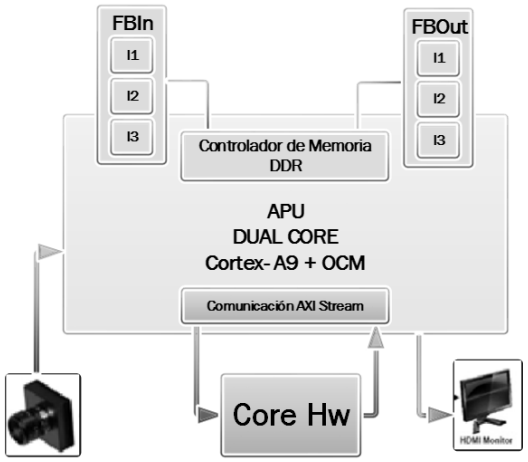


Figura 3.14: Arquitectura de referencia del caso de estudio

Para este caso de estudio, la arquitectura de referencia utilizada está formada por:

- Una cámara conectada por *USB* al microprocesador ARM.
- Dos *framebuffers* para transferencias de imágenes del *core Hw* al microprocesador, y viceversa, por medio de una conexión *VDMA* a un bus *AXI*.
- Un *core Hw* encargado del procesado, que es una instancia del *core* generado en el *Proyecto HLS*.
- Un monitor conectado al microprocesador mediante HDMI.

Además, la arquitectura empleada en este caso de estudio incluye el Sistema Operativo *Linux*. Dicho entorno se encuentra instalado en una tarjeta de memoria *SD* que contiene dos particiones: una con el sistema de archivos con la distribución de *Linux Linaro* y la otra partición con los archivos de arranque y configuración.

Estos archivos de arranque son los siguientes:

- *uImage*: Imagen del *kernel* de Linux. En este caso usamos la versión 3.14.4 (publicada en Mayo de 2014).

- *devicetree.dtb*: Especificación del mapeo de todos los dispositivos del sistema.

- *boot.bin*: Archivo de arranque que contiene la configuración de la *FPGA* y del orden de carga del sistema operativo.

Estos archivos se generan manualmente y es necesario tener una distribución *Linux* para poder generar el **uImage** y la estructura conocida como *devicetree*.

Tercera salida: Proyecto Sw

El tercer proyecto es el *Proyecto Sw*, este proyecto realiza el control del procesado Hw y el resto del procesado de la imagen. La figura 3.15 muestra parte del código Sw que se ejecuta en la plataforma. Este código contiene las instrucciones de configuración y control de los núcleos Hw, y se genera automáticamente. A parte de configurar el núcleo encargado de procesar la imagen, configura el *VDMA* (*Video DMA*) encargado de enviar y recibir los datos procedentes del microprocesador y del Hw.

```
...
AMA_HW_CONF(); //Conf
#pragma amazynq_start:
AMA_HW_START(src, dst); //Ini
#pragma amazynq_stop:
normalize(dst, dst_norm, 0, 255, NORM_MINMAX, CV_32SC1, Mat()
    );
convertScaleAbs(dst_norm, dst_norm_scaled);
...
AMA_HW_STOP();
...
```

Figura 3.15: Detalle de parte del código de salida Sw, *Proyecto Sw*

Igual que en los flujos de datos anterior, *AMAZynqC* genera un grafo dirigido del flujo de datos del código Sw resultante (figura 3.16). Una de los puntos más importantes de este flujo es la ejecución de un hilo encargado de la lectura y escritura de las imágenes en memoria.

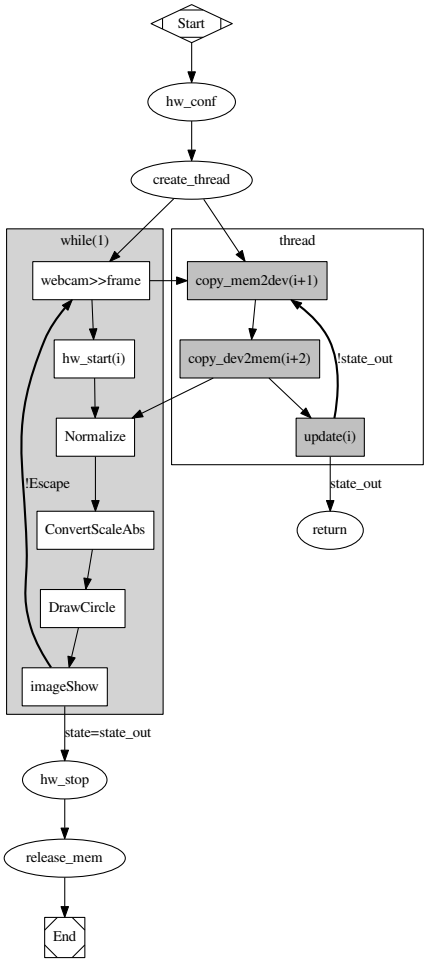


Figura 3.16: Flujo de datos del ejecutable del *Proyecto SW*

Para evitar problemas de lectura y escritura al mismo tiempo entre la función principal, el hilo y el *core HW* se han diseñado dos *framebuffers* (*FBin*, *FBout*) de tamaño igual a tres imágenes, uno para la entrada y otro para la salida del Hw. Debido a esta estructura, se obtiene un desfase en la visualización igual a un *ciclo* más el tiempo en procesar una imagen.

A continuación se describe el funcionamiento del hilo en un ciclo.

1 El hilo empieza escribiendo la nueva imagen obtenida por la cámara en la posición $i + 1$ del *framebuffer* de entrada (*FBin*).

2 El hilo copia la imagen de la posición $i + 2$ del *framebuffer* de salida (*FBout*) y la envía a la función `Normalize()` (imagen de 32 bits entero con signo).

3 Se actualiza la variable i y se comprueba la variable *"state"*, si es igual a `!state_out` termina el hilo.

Por otra parte, el funcionamiento de la función principal es:

1 Se captura la nueva imagen de la cámara (tipo RGB de 8 bits por canal).

2 Se manda la orden *"start"* al Hw para que se procese la imagen de la posición i del *FBin*.

3 Se normaliza la imagen de salida del Hw.

4 Se convierte en valor absoluto.

5 Mediante un umbral de decisión en la detección de esquinas, se pintan los círculos detectados.

6 Se muestra la imagen indefinidamente hasta que no se reciba la orden de terminar.

Al mandar la orden *"terminar"* se cierra el hilo, se vacía la memoria y termina la aplicación.

3.1.2. Test funcional

En este punto del caso de estudio, se va a ejecutar el funcionamiento del sistema en una plataforma *FPGA*. La plataforma de estudio seleccionada es la *Zedboard*, caracterizada por tener un *SoC Zynq* (modelo ZC7020), 512MB de memoria RAM, salida HDMI y otros periféricos.

Con ayuda de la herramienta *EDK* y *SDK* se genera el archivo de configuración de la placa (`system.bit` y *boot.bin*). Además, se obtiene el informe de implementación, ver tabla 3.2, donde se observan los recursos utilizados por el *core Hw*.

Recursos:	**Unidades**
BRAM	23
DSP	31
FF	3917
LUT	4856
SLICE	2139

Tabla 3.2: Informe de recursos del *core Hw* obtenidos con *Xilinx EDK*

Para verificar el sistema, se ejecuta la aplicación generada por el *proyecto Sw* en la plataforma *Zedboard*, donde se comprueba el funcionamiento del sistema (fig 3.17).

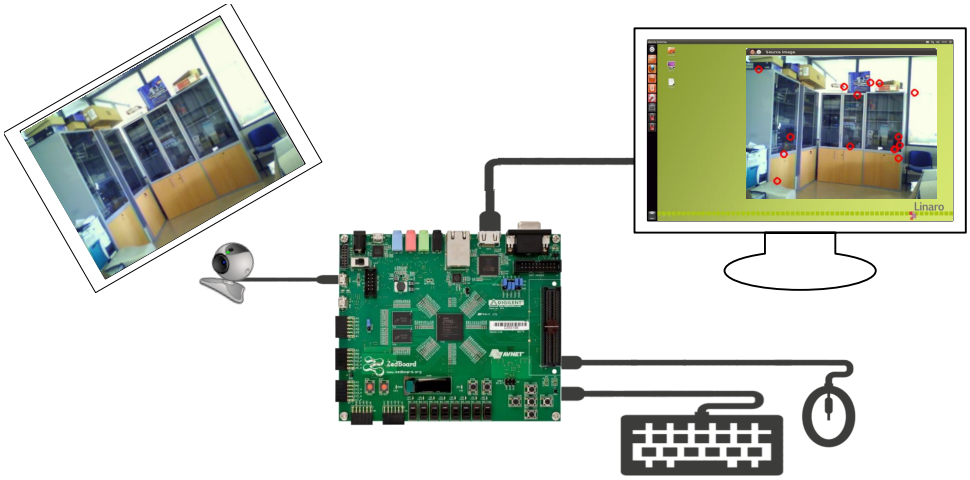


Figura 3.17: Test funcional realizado en la plataforma *Zedboard*

Las imágenes obtenidas representan: la primera una imagen capturada por una cámara *USB* (izquierda) y la segunda la imagen procesada (derecha). Se puede comprobar las esquinas detectados en la imagen por el sistema completo ya que están pintadas de color rojo.

3.2. Caso 2: Exploración del espacio de diseño

En este caso de estudio se plantea realizar una exploración del espació de diseño definido en la región entre los *pragmas* para distintos bloques de procesamiento y diferentes tipos y tamaños de datos.

En la figura 3.18 se observan los *pragmas* que van a definir la región para la exploración. *AMAZynqC* realiza la exploración entre la región de código definida entre `#pragma amazynq_start` y `#pragma amazynq_stop` y, además, intenta definir, si es posible, el tipo de dato indicado por el usuario mediante el *pragma* `#pragma amazynq src_gray.DT = <tipo de dato>`.

```
...
#pragma amazynq_start:  //Pragma Entrada

hls::cvtColor (src, src_gray, CV_BGR2GRAY);
#pragma amazynq src_gray.DT=CV_8UC1:
...
hls::Sobel(gray, dx, depth, 1, 0, 3, scale, 0, BORDER_DEFAULT
    );
#pragma amazynq dx.DT=CV_16SC1:
Sobel(gray, dy, depth, 0, 1, 3, scale, 0, BORDER_DEFAULT);
#pragma amazynq dy.DT=CV_16SC1:
...
calcHarris(Fx, Fy, Fxy, dst, 0.04);
#pragma amazynq dy.DT=CV_32FC1:

#pragma amazynq_stop:  //Pragma Salida
...
```

Figura 3.18: Detalle de los `Pragmas` de la exploración en el espacio de diseño

El primer punto que se explora es el tipo y tamaño de variable que soportan los métodos de la biblioteca *OpenCV* (se han excluido aquellos de 64 bits por píxel y canal). En la tabla 3.3 se observan los tipos soportados.

Type	In	Out
Cam	—	CV_8UC3
cvtColor	CV_8UC3	CV_8UC1
sobel	CV_8U	CV_16S
		CV_32F
BoxFilter	CV_32S	CV_32S
	CV_32F	CV_32F
calcHarris	CV_32S	CV_32F
	CV_32F	CV_32F

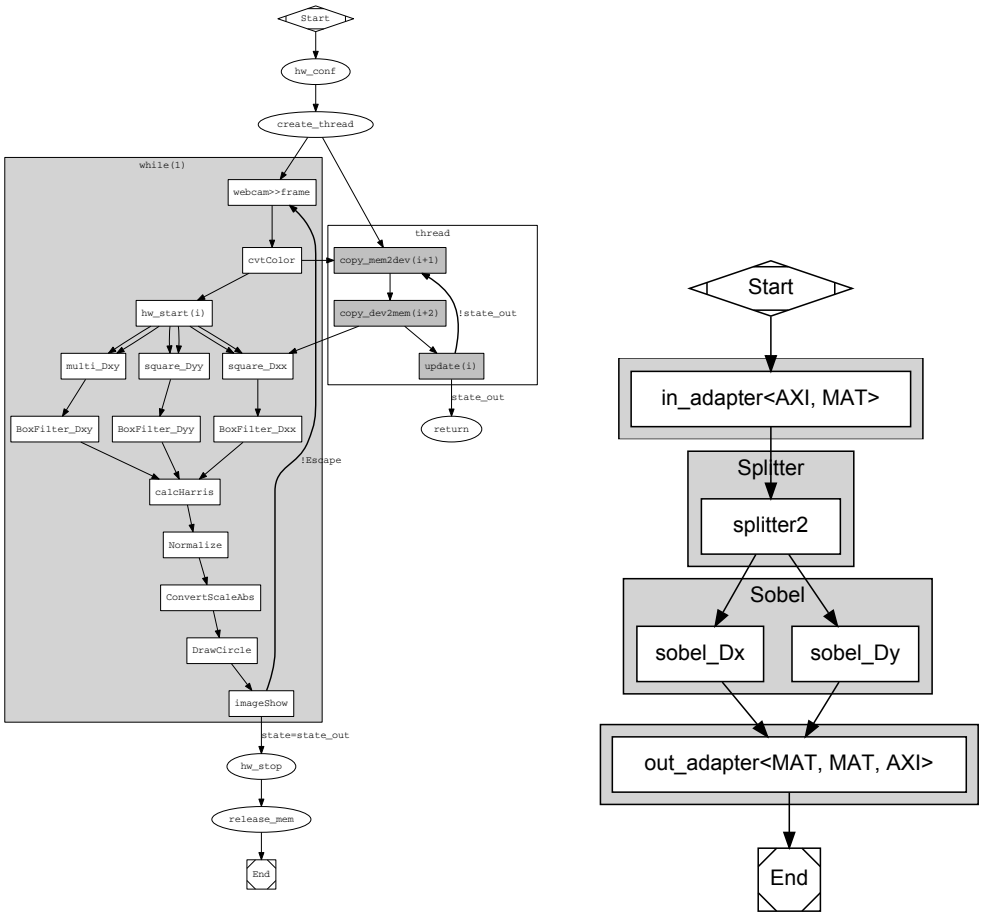
Tabla 3.3: Tipos soportados en este caso de estudio

Estos métodos admiten más posibilidades pero se muestran sólo los que siguen una lógica coherente con el flujo. Por ejemplo, si se multiplica dos valores enteros de 16bits, a la salida se obtiene un valor entero de 32bits, por ello, descartamos los inferiores a 32 bits.

Con la selección del tipo de dato ya realizado (marcado de color azul en la tabla 3.3), sólo falta comprobar qué métodos (consecutivos) son los apropiados para implementarse en Hw. Por ello, se genera una exploración del diseño (generada por el compilador) donde se sintetiza cada tramo del flujo y se comparan los resultados obtenidos.

Igual que en el caso de estudio anterior, *AmazynqC* genera grafos dirigidos descritos en *Graphviz-DOT* del flujo de datos para cada partición (grafo *core HW* y *Sw placa*).

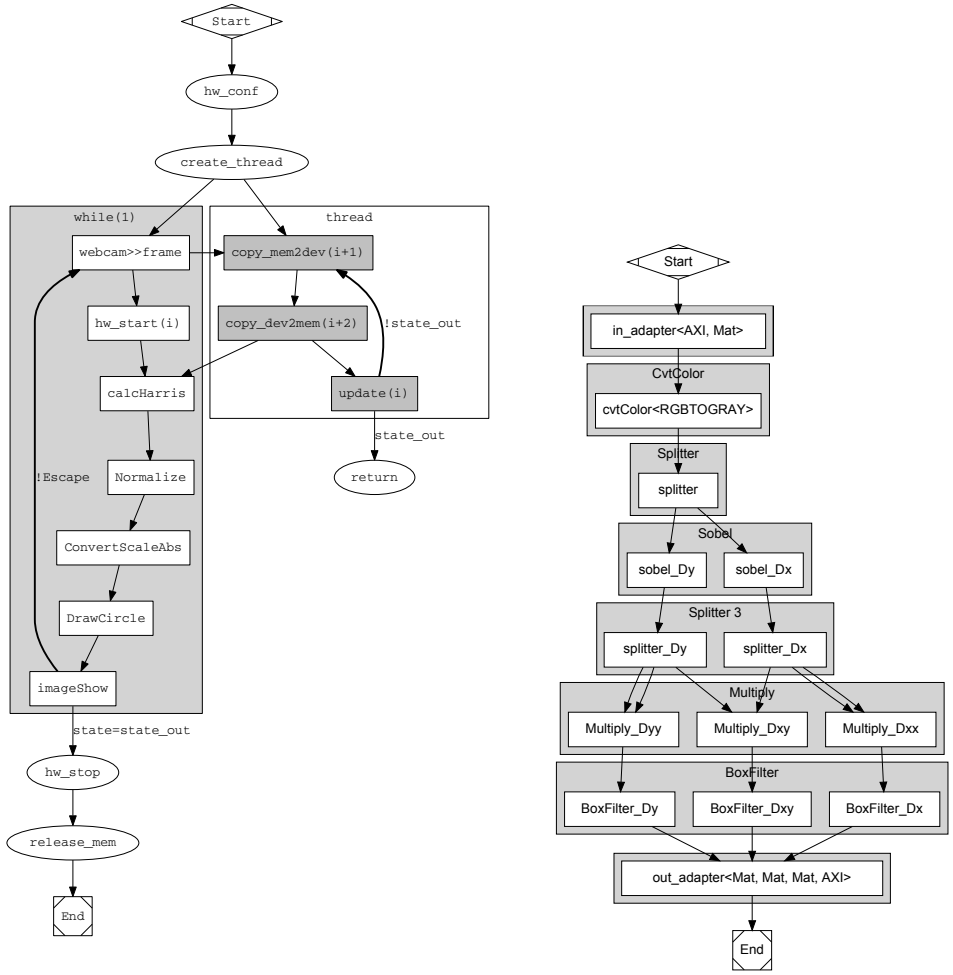
En la figura 3.19 se observa los grafos de los flujos de datos involucrados en el sistema de la partición llamada *Sobel*. Esta partición sólo contiene el clon del método **sobel** en el *core hw*. Debido a esto, casi todo el procesado es realizado por el microprocesador ARM.



(a) Flujo SW (b) Flujo Core Hw

Figura 3.19: Partición Hasta Sobel

Por contra, en la figura 3.20 se observa los grafos de la partición *until_boxfilter*. En este caso, casi todo el procesado se realiza en el core Hw excepto el método *calcHarris*. Este método realiza operaciones de aritmética básica pero, por contra, necesita de tres imágenes de entrada. Debido a esto, el ancho del bus *AXI Stream* de salida es mayor (128bits).



(a) Flujo SW (b) Flujo Core Hw

Figura 3.20: Partición Hasta Sobel

En la tabla 3.4 se observa el informe de síntesis para cada partición realizada que ha sido generado automáticamente por el compilador con la ayuda de la herramienta *Vivado HLS*. El nombre que hemos dado a las particiones sigue el siguiente formato:

Name	BRAM	DSP	FF	LUT	Depth In	Depth Out
Disp. en ZC702	280	220	106400	53200	1024 bits	1024 bits
Hw_all	23	31	8029	10425	32 bits	32 bits
Hw_cvtColor	0	3	526	624	32 bits	32 bits
Hw_from_cvtColor	23	25	7589	9882	32 bits	32 bits
Hw_sobel	6	12	1885	2827	32 bits	32 bits
Hw_until_sobel	6	15	2137	3044	32 bits	32 bits
Hw_from_sobel	20	19	6894	8861	32 bits	32 bits
Hw_multi	0	1	1141	2009	32 bits	128 bits
Hw_until_multi	3	10	2089	3247	32 bits	128 bits
Hw_from_multi	60	18	15791	18302	128 bits	32 bits
Hw_BoxFilter	60	0	13999	15797	128 bits	128 bits
Hw_until_BoxFilter	23	10	5712	6781	32 bits	128 bits
Hw_calcHarris	0	21	3066	4396	128 bits	32 bits

Tabla 3.4: Resultados de Síntesis de cada caso de la partición Hw

- `HW_X`: Sólo la función X está implementada en HW.
- `HW_until_X`: Se implementa en Hw todo el flujo hasta la función X incluida.
- `HW_from_X`: Se implementa en Hw desde la función X en adelante.

Con el informe generado, el usuario solamente tiene que escoger qué partición debe ser implementada en Hw. En este caso de estudio se han calculado 13 posibles candidatos incluyéndose el *ALL_SW* y el *ALL_HW*, empezando desde todo el sistema en Sw hasta todo en Hw (no se muestra en la tabla el informe de todo Sw ya que los recursos lógicos son cero).

También se genera otro informe con distintas métricas de interés para el ejecutable *ARM*:

- `.text`: Tamaño del código.
- `.data`: Espacio reservado para variables inicializadas.
- `.bss`: Espacio reservado para variables no inicializadas.
- `.dec`: La suma de `.text, .bss, .dec`.

En el informe 3.5 no se aprecian unas diferencias considerables. Además, se debe tener el cuenta el tamaño del *FramesBuffer* en los casos Hw ya que no se mencionan en dicho informe.

Name	.text	.data	.bss	.dec
Sw_all	20868	556	284	21708
Hw_all	18728	544	288	19560
Hw_cvtColor	22191	560	288	23039
Hw_from_cvtColor	20068	548	288	20904
Hw_sobel	24616	564	344	25524
Hw_until_sobel	24392	560	344	25296
Hw_from_sobel	23465	556	344	24365
Hw_multi	26263	564	456	27283
Hw_until_multi	25069	556	456	26081
Hw_from_multi	25428	560	456	26444
Hw_BoxFilter	26532	560	512	27604
Hw_until_BoxFilter	24227	548	512	25287
Hw_calcHarris	26846	568	512	27926

Tabla 3.5: Tamaño del ejecutable Sw de cada caso de la partición Hw

El último análisis que se realizó fue la respuesta temporal del sistema y, para ello, se calcularon los *frames por segundo* (*fps*) en cada caso (ver. tabla 3.6). Este cálculo se realizó con una cámara, cofigurada con una resolución de 640 × 480 píxeles, conectada por *USB* al microprocesador, doblándose la tasa de *fps* obtenida respecto a ala partición *ALL_SW*. Esto es debido a que la velocidad de lectura y escritura en memoria *DDR* supone un cuello de botella para la partición Sw.

Name	FPS	Est. Full HD
Sw_all	10	3
Hw_all	20	18
Hw_cvtColor	15	13
Hw_from_cvtColor	19	17
Hw_sobel	15	13
Hw_until_sobel	15	13
Hw_from_sobel	16	14
Hw_multi	14	13
Hw_until_multi	16	14
Hw_from_multi	16	14
Hw_BoxFilter	15	13
Hw_until_BoxFilter	18	16
Hw_calcHarris	15	14

Tabla 3.6: Frames Por Segundo de cada caso de la partición Hw

Por último, también se calculó el tiempo en procesar una imagen *Full HD*

en el caso todo Sw y se estimó la mejora en Hw. Con este caso se obtuvo una ganancia de velocidad de cálculo de $\times 6$.

Por tanto, y como conclusión del caso de estudio, se ha probado que nuestra herramienta *AMAZynq* es capaz de proporcionar distintas particiones Hw/Sw de un sistema empotrado de procesamiento de vídeo, además de distintas métricas de interés para medir las bondades de cada partición. Queda a la elección del diseñador escoger la mejor partición que se adapte a los distintos requisitos involucrados en el diseño.

Capítulo 4

Conclusiones

Este trabajo presenta el entorno *AMAZynq*, una herramienta que valida una nueva metodología para la síntesis e implementación de sistemas empotrados de visión por computador definidos mediante *OpenCV* para las *SoCs* heterogéneos basadas en *Zynq*. Se ha presentado un caso de estudio que valida la herramienta diseñada, explorándose el espacio de diseño mediante distintas particiones Hw/Sw de interés.

El trabajo futuro consiste en seguir aumentando la biblioteca de *cores Hw* con enfoques que acentúen aún más la orientación a la automatización, y que permitan precisar aun más el tipo y tamaño de datos de los distintos flujos involucrados. También se pretende extender el compilador *AMAZynqC* para que calcule los errores cometidos en simulación y mejore las funciones de optimización para ser capaz de optar por la mejor opción sin tener que consultar al usuario.

Bibliografía

[1] Grupo unicad. http://www.dtic.ua.es/unicad/.

[2] NVIDIA Corporation. Jetson TK1 Development Kit. https://developer.nvidia.com/jetson-tk1, 2014.

[3] Preliminary Product Specification. Zynq-7000 All Programmable SoC Overview Zynq-7000 All Programmable SoC First Generation Architecture Processing System (PS) I / O Peripherals and Interfaces Zynq-7000 All Programmable SoC Overview Programmable I / O Blocks, 2013.

[4] Altera. Altera SoCs: When Architecture Matters. http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html, 2013.

[5] Eduardo Gudis, Pullan Lu, David Berends, Kevin Kaighn, Gooitzen van der Wal, Gregory Buchanan, Sek Chai, and Michael Piacentino. An Embedded Vision Services Framework for Heterogeneous Accelerators. *2013 IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 598–603, June 2013.

[6] Jung Sub Kim, Lanping Deng, Prasanth Mangalagiri, Kevin Irick, Kanwaldeep Sobti, Mahmut Kandemir, Vijaykrishnan Narayanan, Chaitali Chakrabarti, Nikos Pitsianis, and Xiaobai Sun. An Automated Framework for Accelerating Numerical Algorithms on Reconfigurable Platforms Using Algorithmic/Architectural Optimization. *IEEE Transactions on Computer*, 58(12):1654–1667, 2009.

[7] Jonathan van de Belt, Paul D. Sutton, and Linda E. Doyle. Accelerating software radio: Iris on the Zynq SoC. In *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 294–295. Ieee, October 2013.

[8] Opencv: Open source computer vision. http://opencv.org/.

[9] Nikolaos Kavvadias and Kostas Masselos. Hardware design space exploration using HercuLeS HLS. In *Proceedings of the 17th Panhellenic*

Conference on Informatics - PCI '13, page 195, New York, New York, USA, 2013. ACM Press.

[10] Joachim Keinert, Martin Streubuhr, Thomas Schlichter, Joachim Falk, Jens Gladigau, Christian Haubelt, Jurgen Teich, and Michael Meredith. SystemCoDesigner—An automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–23, January 2009.

[11] Vivado high-level synthesis. http://www.xilinx.com/products/design-tools/vivado/index.htm.

[12] Christian Pilato, Riccardo Cattaneo, Gianluca Durelli, Alessandro Antonio Nacci, Marco Domenico Santambrogio, Donatella Sciuto, and Milano Deib. A2B : an Integrated Framework for Designing Heterogeneous and Reconfigurable Systems. In *2013 NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2013)*, pages 198–205, 2013.

[13] Antonio Filgueras, Eduard Gil, Daniel Jimenez-Gonzalez, Carlos Alvarez, Xavier Martorell, Jan Langer, Juanjo Noguera, and Kees Vissers. OmpSs@Zynq all-programmable SoC ecosystem. *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays - FPGA '14*, pages 137–146, 2014.

[14] Antonio Filgueras, Eduard Gil, Carlos Alvarez, Daniel Ji, Xavier Martorell, and Jan Langer. Heterogeneous Tasking on SMP / FPGA SoCs: the Case of OmpSs and the Zynq. In *2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC)*, 2013.

[15] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp : An Open Source High-Level Synthesis Tool for FPGA-Based Processor / Accelerator Systems. *ACM Transactions on Embedded Computing Systems*, 1(1), 2012.

[16] Shakith Fernando, Firew Siyoum, Yifan He, Akash Kumar, and Henk Corporaal. MAMPSx: A design framework for rapid synthesis of predictable heterogeneous MPSoCs. *2013 International Symposium on Rapid System Prototyping (RSP)*, (1):136–142, October 2013.

[17] Axi reference guide. http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[18] Xilinx. Xilinx UG925 Zynq-7000 All Programmable SoC ZC702 Base Targeted Reference Design User Guide (ISE Design Suite 14.5). Technical report, 2013.

[19] Linux with hdmi video output on the zed and zc702, zc706 boards). http://wiki.analog.com/resources/tools-software/linux-drivers/platforms/zynq.

[20] Harris Corner Detector. http://docs.opencv.org/doc/tutorials/features2d/trackingmotion/harris_detector/harris_detector.html.

[21] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK, 1988.

[22] Pei-Yung Hsiao, Chieh-Lun Lu, and Li-Chen Fu. Multilayered image processing for multiscale harris corner detection in digital realization. *Industrial Electronics, IEEE Transactions on*, 57(5):1799–1805, 2010.

[23] J.-B. Ryu, C.-G. Lee, and H.-H. Park. Formula for harris corner detector. *Electronics Letters*, 47(3):180–181, 2011.